Claudio De Sio Cesari

Manuale di Java 9

Programmazione orientata agli oggetti con Java Standard Edition 9

- Moduli: Java non sarà mai più lo stesso!
- Java interattivo con JShell
- Impara a programmare partendo da zero

HOEPLI
INFORMATICA

MANUALE DI JAVA 9

Claudio De Sio Cesari

MANUALE DI JAVA 9

Programmazione orientata agli oggetti con Java Standard Edition 9

EDITORE ULRICO HOEPLI MILANO

Copyright © Ulrico Hoepli Editore S.p.A. 2018
via Hoepli 5, 20121 Milano (Italy)
tel. +39 02 864871 – fax +39 02 8052886
e-mail hoepli@hoepli.it

www.hoepli.it

Seguici su Twitter: @Hoepli_1870

www.hoepli.it

Tutti i diritti sono riservati a norma di legge
e a norma delle convenzioni internazionali

ISBN EBOOK 978-88-203-8302-2

Progetto editoriale: Emanuele Giuliani (emanuele@giuliani.mi.it)
Copertina: Sara Taglialegne

Realizzazione digitale: Promedia, Torino

Indice generale

Prefazione

Parte I - Le basi del linguaggio

Capitolo 1 - Introduzione a Java

1.1 Prerequisiti: concetti informatici di base
- 1.1.1 Hardware
- 1.1.2 Software
- 1.1.3 Linguaggi di programmazione

1.2 Introduzione a Java
- 1.2.1 Cos’è Java
- 1.2.2 Caratteristiche del linguaggio Java

1.3 Ambiente di sviluppo
- 1.3.1 Struttura del JDK
- 1.3.2 Guida per sviluppare passo dopo passo

1.4 Primo approccio al codice
- 1.4.1 Analisi del programma HelloWorld
- 1.4.2 Compilazione ed esecuzione del programma HelloWorld
- 1.4.3 La nostra interazione con il computer

1.5 Possibili problemi in fase di compilazione ed esecuzione
- 1.5.1 Possibili messaggi di errore in fase di compilazione
- 1.5.2 Possibili messaggi relativi alla fase di interpretazione

1.6 Altri ambienti di sviluppo

Riepilogo

Capitolo 2 - Componenti fondamentali di un programma Java

2.1 Cosa significa creare un programma Java
- 2.1.1 Programmazione e paradigmi
- 2.1.2 Le basi della programmazione ad oggetti (OOP)

2.2 Le basi della programmazione object oriented: classi ed oggetti
- 2.2.1 Osservazione importante sulla classe `Punto`
- 2.2.2 Osservazione importante sulla classe `TestOggettiPunto`
- 2.2.3 Un'altra osservazione importante

2.3 I metodi in Java
- 2.3.1 Dichiarazione di un metodo
- 2.3.2 Chiamata (o invocazione) di un metodo
- 2.3.3 Varargs

2.4 Le variabili in Java
- 2.4.1 Dichiarazione di una variabile
- 2.4.2 Variabili d'istanza
- 2.4.3 Variabili locali
- 2.4.4 Parametri formali

2.5 I metodi costruttori
- 2.5.1 Caratteristiche di un costruttore
- 2.5.2 Costruttore di default

2.6 Introduzione ai package
- 2.6.1 Package e import
- 2.6.2 Gestione dei package manuale

2.7 Altri componenti fondamentali della programmazione Java
- 2.7.1 Interfacce
- 2.7.2 Enumerazioni
- 2.7.3 Annotazioni

2.7.4 Moduli
2.7.5 Inizializzatori
2.7.6 Classi innestate
2.7.7 Espressioni lambda

Riepilogo

Capitolo 3 - Stile di codifica, tipi di dati ed array

3.1 Stile di codifica
3.1.1 Schema Libero
3.1.2 Commenti
3.1.3 Regole per gli identificatori
3.1.4 Standard e convenzioni per i nomi

3.2 Gestione della memoria
3.2.1 Byte
3.2.2 Sistemi numerici
3.2.3 Complemento a due

3.3 Tipi di dati primitivi
3.3.1 Tipi di dati interi
3.3.1.1 Promozione automatica nelle espressioni
3.3.1.2 Casting
3.3.1.3 Operazioni con interi
3.3.2 Tipi di dati a virgola mobile, casting e promotion
3.3.3 Underscore in tipi di dati numerici
3.3.4 Tipo di dato logico - booleano
3.3.5 Tipo di dato primitivo letterale

3.4 Tipi di dati non primitivi: reference
3.4.1 Passaggio di parametri per valore
3.4.2 Inizializzazione delle variabili d’istanza

3.5 Introduzione alla libreria standard
3.5.1 Il comando `import`
3.5.2 La classe `String`
3.5.3 La documentazione della libreria standard di Java

3.5.4 Lo strumento javadoc

3.6 Gli array in Java

3.6.1 Dichiarazione
3.6.2 Creazione
3.6.3 Inizializzazione
3.6.4 Array Multidimensionali
3.6.5 String `args[]`

Riepilogo

Capitolo 4 - Operatori e Gestione del flusso di esecuzione

4.1 Operatori di base

4.1.1 Operatore d’assegnazione
4.1.2 Operatori aritmetici
4.1.3 Operatori (unari) di pre e post-incremento (e decremento)
4.1.4 Operatori bitwise
4.1.5 Operatori relazionali o di confronto
4.1.6 Operatori logico - booleani
4.1.7 Concatenazione di stringhe con `+`
4.1.8 Priorità degli operatori

4.2 Costrutti di programmazione semplici

4.2.1 Il costrutto `if`
4.2.2 Il costrutto `while`

4.3 Costrutti di programmazione avanzati

4.3.1 Il costrutto `for`
4.3.2 Il costrutto `do`
4.3.3 Ciclo for migliorato
4.3.4 Il costrutto `switch`
4.3.5 L’operatore ternario
4.3.6 `break` e `continue`

4.4 Programmare meglio

4.4.1 Approccio alla programmazione
4.4.2 Algoritmo
4.4.3 Introduzione ad UML

Riepilogo

Capitolo 5 - Sviluppare realmente con Java

5.1 JShell
5.1.1 HelloWorld con JShell
5.1.2 JShell e codice Java
5.1.2.1 Esempi pratici
5.1.2.2 Regole JShell
5.1.3 Comandi JShell
5.1.3.1 Comandi esplorativi
5.1.3.2 Comandi esecutivi
5.1.4 Strumenti ausiliari di JShell
5.1.4.1 Variabili implicite
5.1.4.2 Forwarding reference
5.1.4.3 Auto-completamento
5.1.4.4 Il comando **/edit**
5.1.4.5 Scorciatoie da tastiera (shortcuts)

5.2 Integrated Development Environment (IDE)
5.2.1 Progetto
5.2.2 Editor
5.2.3 Debugger
5.2.4 Integrazioni

5.3 Realtà aziendale
5.3.1 Architettura
5.3.2 Conoscenze
5.3.3 Tipologie di programmi
5.3.4 Ruoli
5.3.5 Metodologia

5.4 Caso di studio

5.4.1 Raccolta requisiti con Use case e scenari
5.4.1.1 Use case diagram
5.4.1.2 Scenari dei casi d’uso
5.4.2 Architettura di alto livello
5.4.3 Key abstraction
5.4.4 Progettazione con diagrammi di interazione

Riepilogo

Parte II - Object Orientation

Capitolo 6 - Incapsulamento e visibilità

6.1 Breve storia della programmazione ad oggetti

6.2 I paradigmi della programmazione ad oggetti
6.2.1 Astrazione
6.2.2 Riuso
6.2.3 UML: Diagramma delle classi

6.3 Incapsulamento
6.3.1 Incapsulamento funzionale
6.3.2 Incapsulamento e riuso
6.3.3 Il reference `this`
6.3.4 Uso di `this` con variabili
6.3.5 Uso di `this` con metodi e costruttori

6.4 Quando utilizzare l’incapsulamento

6.5 Come usare l’incapsulamento

6.6 Gestione dei package
6.6.1 Gestione manuale
6.6.2 Moduli

6.7 Modificatori d’accesso
6.7.1 Modificatore `public`

6.7.2 Modificatore `protected`
6.7.3 Nessun modificatore d'accesso
6.7.4 Modificatore `private`

6.8 Il modificatore `static`
6.8.1 Metodi statici
6.8.2 Variabili statiche (di classe)
6.8.3 Inizializzatori statici ed inizializzatori d'istanza
6.8.4 Static import
6.8.5 Quando usare `static`
6.8.6 Design Pattern Singleton

Riepilogo

Capitolo 7 - Ereditarietà ed interfacce

7.1 Ereditarietà
7.1.1 La parola chiave `extends`

7.2 Il modificatore `final`

7.3 La classe `Object`

7.4 Rapporto ereditarietà-incapsulamento

7.5 Quando utilizzare l'ereditarietà
7.5.1 La relazione "is a"
7.5.2 Ereditarietà e costruttori
7.5.3 La parola chiave `super`
7.5.4 Ereditarietà e modificatori
7.5.5 Ereditarietà e inizializzatori
7.5.6 Generalizzazione e specializzazione

7.6 Il modificatore `abstract`
7.6.1 Metodi astratti
7.6.2 Classi astratte

7.7 Interfacce
7.7.1 Definizione classica (pre-Java 8)

7.7.2 Metodi statici
7.7.3 Metodi di default
7.7.4 Interfacce funzionali
7.7.5 Ereditarietà multipla
7.7.5.1 Diamond Problem
7.7.5.2 Conflitto tra un metodo astratto e uno concreto
7.7.5.3 Conflitto tra due metodi astratti
7.7.5.4 Conflitto tra metodi ridefiniti
7.7.5.5 Class always wins
7.7.6 Differenze tra interfacce e classi astratte

Riepilogo

Capitolo 8 - Polimorfismo

8.1 Polimorfismo
8.1.1 Convenzione per i reference

8.2 Polimorfismo per metodi
8.2.1 Overload
8.2.2 Varargs
8.2.3 Override
8.2.4 Annotazione sull'override

8.3 Polimorfismo per dati
8.3.1 Parametri polimorfi
8.3.2 Collezioni eterogenee
8.3.3 Casting di oggetti
8.3.4 Invocazione virtuale dei metodi
8.3.5 Esempio d'utilizzo del polimorfismo
8.3.6 Polimorfismo e interfacce

Riepilogo

Capitolo 9 - Eccezioni ed asserzioni

9.1 Eccezioni, errori ed asserzioni

9.2 Gerarchie e categorizzazioni

9.3 Meccanismo per la gestione delle eccezioni

9.4 Try with resources

9.5 Eccezioni personalizzate e propagazione dell'eccezione
- 9.5.1 Warnings
- 9.5.2 Altra regola sull'override

9.6 Introduzione alle asserzioni
- 9.6.1 Progettazione per contratto
- 9.6.2 Note per l'esecuzione di programmi che utilizzano la parola `assert`
- 9.6.3 Quando usare le asserzioni
  - 9.6.3.1 Consiglio 1
  - 9.6.3.2 Consiglio 2
  - 9.6.3.3 Consiglio 3
- 9.6.4 Conclusioni

Riepilogo

Conclusioni Parte II

Parte III - Caratteristiche avanzate

Capitolo 10 - Enumerazioni e Tipi Innestati

10.1 Classi innestate: classi interne
- 10.1.1 Classe innestata: definizione
- 10.1.2 Classi innestate: proprietà
- 10.1.3 Quando usare le classi innestate

10.2 Classi anonime: definizione
- 10.2.1 Quando usare le classi innestate

10.3 Tipi Enumerazioni
- 10.3.1 Ereditarietà e polimorfismo con `enum`

10.3.2 Metodi, variabili, costruttori e tipi innestati in una enumerazione
10.3.3 Quando utilizzare una `enum`
10.3.4 Enumerazioni innestate (in classi) o enumerazioni membro
10.3.5 Enumerazioni e metodi specifici degli elementi
10.3.6 `Switch` e `enum`

Riepilogo

Capitolo 11 - Generics

11.1 Generics e tipi parametro
11.1.1 Generics e collection
11.1.2 Tipi primitivi e autoboxing-autounboxing
11.1.3 Interfaccia `Iterator`
11.1.4 Interfaccia `Map`
11.1.5 Deduzione automatica del tipo
11.1.6 Raw type e XLint

11.2 Ereditarietà e tipi generici
11.2.1 Ereditarietà e tipo parametro
11.2.2 Type erasure
11.2.3 Wildcard

11.3 Creare propri tipi generici
11.3.1 Parametri bounded
11.3.2 Wildcard bounded
11.3.3 Metodi generici
11.3.4 Wildcard capture e metodi helper

11.4 Parametri covarianti
11.4.1 Problema dei parametri covarianti
11.4.2 Soluzione con bounded parameter

Riepilogo

Capitolo 12 - La libreria indispensabile: il package `java.lang`

12.1 La classe `Object`
- 12.1.1 Il metodo `equals()`
- 12.1.2 Il metodo `hashcode()`
- 12.1.3 Il metodo `toString()`
- 12.1.4 Il metodo `clone()`

12.2 La classe `System`
- 12.2.1 Input e output
- 12.2.2 Le properties
- 12.2.3 Altri metodi di `System`
- 12.2.4 Garbage collection

12.3 La classe `Runtime`

12.4 La classe `Class` e `Reflection`
- 12.4.1 Istanziare un oggetto `Class`
- 12.4.2 `Class` è un tipo generico
- 12.4.3 Un esempio di reflection
- 12.4.4 Nomi dei parametri dei metodi

12.5 La classe `String`
- 12.5.1 Gestione in memoria
- 12.5.2 Dichiarazione della classe `String`
- 12.5.3 I metodi di `String`
  - 12.5.3.1 `char charAt(int index)`
  - 12.5.3.2 `String concat(String other)`
  - 12.5.3.3 `int compareTo(String other)`
  - 12.5.3.4 `boolean endsWith(String suffix)`
  - 12.5.3.5 `boolean equals(Object ob)`
  - 12.5.3.6 `boolean equalsIgnoreCase(String s)`
  - 12.5.3.7 `static String format(String format, Object... args)`
  - 12.5.3.8 `int indexOf(int ch)`
  - 12.5.3.9 `int indexOf(int ch, int fromIndex)`
  - 12.5.3.10 `String intern()`
  - 12.5.3.11 `static String join(CharSequence delimiter, CharSequence... elements)`
  - 12.5.3.12 `static join(CharSequence delimiter, Iterable<?`

extends CharSequence> elements)
12.5.3.13 int lastIndexOf(int ch)
12.5.3.14 int length()
12.5.3.15 String replace(char oldChar, char newChar)
12.5.3.16 String replace(CharSequence target, CharSequence replacement)
12.5.3.17 String replaceAll(String regex, String replacement)
12.5.3.18 String replaceFirst(String regex, String replacement)
12.5.3.19 String[] split(String regex)
12.5.3.20 boolean startsWith(String prefix)
12.5.3.21 boolean startsWith(String prefix, int fromIndex)
12.5.3.22 String substring(int startIndex)
12.5.3.23 String substring(int startIndex, int endIndex)
12.5.3.24 String toLowerCase()
12.5.3.25 String toString()
12.5.3.26 String toUpperCase()
12.5.3.27 String trim()
12.5.3.28 static String valueOf(Object obj)
12.5.4 Compact Strings

12.6 Le classi wrapper
12.6.1 Autoboxing-Autounboxing
12.6.1.1 Gestione in memoria dei tipi primitivi
12.6.1.2 Assegnazione di un valore null al tipo wrapper
12.6.1.3 Costrutti del linguaggio ed operatori relazionali
12.6.1.4 Overload
12.6.2 Altri metodi delle classi wrapper

12.7 La classe Math

12.8 Ordinamento
12.8.1 L'interfaccia Comparable
12.8.2 L'interfaccia Comparator

Riepilogo

Capitolo 13 - Tipi Annotazioni

13.1 Definizione di annotazione (metadato)

13.1.1 Primo esempio

13.1.2 Tipologie di annotazioni e sintassi

13.1.2.1 Annotazione ordinaria (o completa)

13.1.2.2 Annotazione a valore unico

13.1.2.3 Annotazione marcatrice

13.2 Annotare annotazioni (meta-annotazioni)

13.2.1 `Target`

13.2.1.1 Type Annotations

13.2.2 `Retention`

13.2.3 `Documented`

13.2.4 `Inherited`

13.2.5 `Repeatable`

13.3 Annotazioni standard

13.3.1 `Override`

13.3.2 `Deprecated`

13.3.3 `FunctionalInterface`

13.3.4 `SuppressWarnings`

13.3.5 `Native`

Riepilogo

Capitolo 14 - Le librerie di utilità: il package `java.util` e Date-Time API

14.1 Il package `java.util`

14.1.1 Le classi `Properties` e `Preferences`

14.1.2 Internazionalizzazione (i18n)

14.1.2.1 Classe `Locale` ed internazionalizzazione

14.1.2.2 La classe `ResourceBundle`

14.1.3 La classe `StringTokenizer`

14.1.4 Formattazioni di output

14.1.5 Espressioni regolari
14.1.6 Date, orari e valute

14.2 La novità di Java 8: Date-Time API
14.2.1 Standardizzazione dei metodi
14.2.2 Il package `java.time`
14.2.2.1 La classe `Instant`
14.2.2.2 Le classi `Duration e Period`
14.2.2.3 Le enumerazioni `DayOfWeek` e `Month`
14.2.2.4 Date classes: `LocalDate, YearMonth, MonthYear` e `Year`
14.2.2.5 Le classi `LocalTime` e `LocalDateTime`
14.2.2.6 Geolocalizzazione: le classi `ZonedDateTime, ZoneId` e `ZoneOffSet`
14.2.3 Il package `java.time.format`
14.2.4 Il package `java.time.temporal`
14.2.4.1 L'enumerazione `ChronoUnit`
14.2.4.2 Temporal Adjusters
14.2.4.3 Temporal Queries
14.2.5 Codice Legacy

Riepilogo

Capitolo 15 - Gestione dei thread

15.1 Introduzione ai thread
15.1.1 Definizione provvisoria di thread
15.1.2 Cosa significa multithreading

15.2 La classe `Thread` e la dimensione temporale
15.2.1 Analisi di `ThreadExists`
15.2.2 L'interfaccia `Runnable` e la creazione dei thread
15.2.3 Analisi di `ThreadCreation`
15.2.4 La classe `Thread` e la creazione dei thread

15.3 Priorità, scheduler e sistemi operativi
15.3.1 Analisi di `ThreadRace`
15.3.2 Comportamento Windows (e la maggior parte dei sistemi Linux)

15.3.3 Comportamento Unix
15.3.4 Il modificatore `volatile`

15.4 Thread e sincronizzazione
15.4.1 Analisi di `Synch`
15.4.2 Monitor e lock

15.5 La comunicazione fra thread
15.5.1 Analisi di `IdealEconomy`

15.6 Concorrenza
15.6.1 Oggetti Immutabili
15.6.2 Tecnologie Java e classi standard ausiliarie
15.6.3 Libreria Java sulla concorrenza
15.6.3.1 Il package `java.util.concurrent.locks`
15.6.3.2 Il package `java.util.concurrent.atomic`
15.6.3.3 Interfacce Executors
15.6.3.4 La classe `Semaphore`
15.6.3.5 La classe `CyclicBarrier`

Riepilogo

Capitolo 16 - Espressioni Lambda

16.1 Espressioni lambda
16.1.1 Sintassi
16.1.2 Comprendere le espressioni lambda
16.1.3 Espressione Lambda vs Classe Anonima
16.1.3.1 Sinteticità
16.1.3.2 Regole e visibilità
16.1.3.3 Dinamicità
16.1.3.4 Gestione delle eccezioni
16.1.3.5 Quando usare le espressioni lambda

16.2 Reference a metodi
16.2.1 Sintassi
16.2.2 Reference a un metodo statico
16.2.3 Reference a un metodo d’istanza

16.2.4 Reference a un metodo d’istanza di un certo tipo
16.2.5 Reference a un costruttore

16.3 Le interfacce funzionali del package `java.util.function`
16.3.1 `Predicate`
16.3.2 `Consumer`
16.3.3 `Supplier`
16.3.4 `Function`
16.3.5 `UnaryOperator` e composizione di espressioni lambda

Riepilogo

Capitolo 17 - Collections Framework e Stream API

17.1 Introduzione al Framework Collections

17.2 L’interfaccia `Collection`
17.2.1 Metodi di `Collection`
17.2.2 Iterare sulle collezioni
17.2.2.1 Ciclo foreach
17.2.2.2 Iteratori
17.2.2.3 Metodo `forEach()` dell’interfaccia `Iterable`

17.3 Interfaccia `List`
17.3.1 Implementazioni di `List`

17.4 Le interfacce `Set` e `SortedSet`
17.4.1 Implementazioni di `Set` e `SortedSet`

17.5 Le interfacce `Queue` e `Deque`
17.5.1 L’interfaccia `BlockingQueue` e implementazioni di `Queue` e `Deque`

17.6 Le interfacce `Map` e `SortedMap`
17.6.1 Implementazioni di `Map` e `SortedMap`

17.7 Algoritmi e utilità
17.7.1 `Collections`
17.7.1.1 Metodi wrapper
17.7.1.2 Metodi sincronizzatori

17.7.1.3 Metodi di convenienza
17.7.2 `Arrays`
17.7.3 Metodi di convenienza per creare collection immutabili
17.7.3.1 Interfaccia `Set`
17.7.3.2 Interfaccia `Map`
17.7.3.3 Interfaccia `List`

17.8 Introduzione alla libreria Stream API
17.8.1 Definizioni di `Stream` e pipeline
17.8.2 Classi Optional
17.8.2.1 Altri metodi `or()`
17.8.2.2 Metodi `get()`, `filter()`, `map()` e `flatMap()`
17.8.3 Reduction Operations
17.8.3.1 Metodo `reduce()`
17.8.3.2 Metodo `collect()` e la classe `Collectors`
17.8.4 Stream paralleli e prestazioni

Riepilogo

Capitolo 18 - Input-Output

18.1 Introduzione all'input-output

18.2 Pattern Decorator
18.2.1 Descrizione del pattern
18.2.2 Esempio

18.3 Caratteristiche del package `java.io`
18.3.1 I Character Stream
18.3.2 I Byte Stream
18.3.3 Le interfacce alla base della gerarchia
18.3.4 Checked exception e chiusura degli stream

18.4 Input e output classici
18.4.1 Lettura di input da tastiera
18.4.2 Gestione dei file
18.4.3 Serializzazione di oggetti e modificatore `transient`
18.4.4 Deep copy

18.5 NIO 2
18.5.1 L’interfaccia `Path`
18.5.2 La classe `Files`

Riepilogo

Capitolo 19 - Moduli

19.1 Perché un sistema modulare?
19.1.1 Pro
19.1.2 Contro

19.2 Definizione di modulo
19.2.1 `HelloModularWorld`
19.2.2 Nozioni architetturali e di progettazione
19.2.3 Direttive
19.2.3.1 `requires`
19.2.3.2 `exports`
19.2.3.3 `opens` e `open` (forte incapsulamento)
19.2.3.4 `uses e provides to`

19.3 Servizi con ServiceLoader
19.3.1 Pattern Factory
19.3.2 Factory modulare
19.3.3 Evoluzione con ServiceLoader
19.3.4 Compilazione ed esecuzione
19.3.5 Metodo `provider`

19.4 Jar modulare
19.4.1 Strumento jar
19.4.2 Formato jmod

19.5 Migrazione di programmi pre-Java 9
19.5.1 Modulo anonimo
19.5.2 Modulo automatico

19.6 Jlink

Riepilogo

Conclusioni

Indice analitico

Informazioni sul Libro

Circa l’autore

Prefazione

Manuale di Java 9 è strutturato in modo tale da facilitare l'apprendimento del linguaggio, anche a chi non ha mai programmato, o a chi ha programmato con linguaggi funzionali. Infatti, la forma espositiva, la struttura ed i contenuti del testo, sono stati accuratamente scelti basandosi sull'esperienza che ho accumulato come formatore e mentore in circa vent'anni di carriera. In particolare, per Sun Educational Services Italia (ora Oracle), ho avuto la possibilità di erogare corsi per migliaia di discenti su tecnologia Java, architettura, analisi e progettazione object oriented ed UML.

Questo testo, è stato creato sulle ceneri di altri due manuali su Java di natura free: "Il linguaggio Object Oriented Java" e "Object Oriented && Java 5". Entrambi questi manuali sono stati scaricati decine di migliaia di volte da http://www.claudiodesio.com e da molti importanti mirror specializzati nel settore, riscuotendo tantissimi consensi da parte dei lettori. In particolare "Object Oriented && Java 5" è stato scaricato oltre 400000 volte! Inoltre anche le precedenti versioni cartacee di questo libro ("Manuale di Java 6", "Manuale di Java 7" e soprattutto "Manuale di Java 8" sempre editi da Hoepli) hanno riscosso un grande successo, e sono stati utilizzati come testi di riferimento in tutte le più importanti università italiane per corsi relativi alla programmazione. Il libro che avete tra le mani è quindi il risultato di un lavoro di anni, basato non soltanto sulla sola esperienza pratica e didattica, ma anche su centinaia di feedback di lettori.

Cosa c'è di nuovo

La versione 9 porta con sé poche modifiche significative al linguaggio con il progetto denominato *Milling Project Coin*. Tra queste ricordiamo l'introduzione dei metodi privati nelle interfacce, e l'impossibilità di utilizzo dell'underscore come identificatore. Ma c'è una grande novità in Java 9: il *Java Platform Module System*. Con l'introduzione del concetto di modulo, il nostro modo di concepire le applicazioni Java e il ciclo di sviluppo del software cambieranno totalmente. Adesso ogni applicazione deve appartenere

almeno ad un modulo, e applicazioni complesse potranno essere partizionate in sottosistemi indipendenti e riutilizzabili rappresentati da moduli. I moduli ci offrono un controllo totale sull'accesso ai tipi di Java, ed è possibile rendere completamente inaccessibili alcuni tipi con il concetto di *forte incapsulamento*. Persino le librerie della piattaforma, tradizionalmente divise in package, sono state reingegnerizzate per essere organizzate in moduli. Per supportare la modularizzazione della piattaforma, sono stati introdotti nuovi formati di impacchettamento come i file JAR modulari e i file JMOD. Soprattutto gli strumenti del JDK esistenti sono stati rivoluzionati e ne sono stati introdotti di nuovi come jmod e soprattutto jlink. Quest'ultimo permette di creare runtime personalizzati che includono solamente le librerie della piattaforma Java che sono necessarie alla nostra applicazione. Un altro strumento fondamentale introdotto è jshell, che ci consente di interagire con il codice Java senza dover creare per forza classi, metodi, e così via.

Struttura del testo

Manuale di Java 9 è diviso in tre parti. Nella prima parte denominata "Le basi di Java", verranno presentati i concetti che permetteranno anche a chi inizia da zero, di cominciare a programmare. Nella seconda parte "Object Orientation" saranno spiegati i concetti fondamentali per progettare correttamente i nostri programmi. Nella terza parte "Caratteristiche avanzate" verranno introdotti e approfonditi tutti gli argomenti più complessi. Si tratta quindi di un testo che dovrebbe soddisfare le aspettative sia dell'aspirante programmatore che del programmatore esperto. Infatti nella prima parte (composta da cinque capitoli), nulla sarà dato per scontato, e verranno introdotte anche nozioni di informatica di base. Inoltre il lettore sarà supportato con esempi, esercizi e spiegazioni semplificate. Ho anche creato un semplice tool di sviluppo gratuito ed open source di nome EJE, per supportare le fasi di apprendimento iniziali. Lo studio dei primi cinque capitoli, dovrebbe quindi rendere il lettore già in grado di scrivere i primi programmi, e anche di avere una certa confidenza con l'ambiente di sviluppo. Nei capitoli della seconda parte ("Object Orientation") sono trattati alcuni argomenti che difficilmente si trovano su altri testi similari. I capitoli che vanno dal sei al nove pongono particolare enfasi sul supporto che il linguaggio offre all'Object Orientation. Solitamente infatti, la difficoltà

maggiore che incontra un programmatore Java, è nel riuscire a sfruttare in pratica proprio i paradigmi della programmazione ad oggetti. Questo testo si sforza quindi di fornire tutte le informazioni necessarie al lettore, per intraprendere la strada della programmazione Java, nel modo più corretto possibile, ovvero in maniera object oriented. La terza parte ("Caratteristiche Avanzate") che comprende i capitoli dal decimo al diciannovesimo, tratta tutte le caratteristiche avanzate del linguaggio, e il grado di approfondimento è particolarmente elevato per supportare anche un'eventuale certificazione Oracle. Si imparerà ad affiancare alla programmazione object oriented, altri tipi di programmazione come quella generica, quella basata sui metadati, quella basata sui contratti, quella multithreaded e quella funzionale.

Inoltre, per non gravare troppo sulla corposità e sul costo dell'opera, moltissimo materiale (21 appendici e centinaia di esercizi) è stato spostato in un apposito spazio disponibile on line agli indirizzi http://www.hoeplieditore.it/8302-2 e http://www.claudiodesio.com/java9.html.

Stili del testo

Il testo è stilizzato in modo da catturare l'attenzione, evidenziando parole, frasi o interi paragrafi. In particolare: con lo stile *Corsivo* sono messi in evidenza termini particolari o di una certa importanza; lo stile **Grassetto** evidenzia parole chiave, concetti importanti del linguaggio Java, nomi di tecnologie o altro ancora a cui l'autore ha voluto dare importanza e su cui richiamare l'attenzione; lo stile di Interfaccia è utilizzato, all'interno del testo, per i link, i comandi e tutto quanto possa essere digitato nelle interfacce di un sistema operativo; lo stile `Codice nel testo` è stato utilizzato per diversificare parti di codice nelle spiegazioni, rispetto al resto del testo. I listati di codice sono inseriti in blocchi evidenziati simili al seguente:

```
public class LaMiaPrimaClasseJava9 {
        //Questo è un listato di codice Java 9!
}
```

Mentre gli input da riga di comando e gli output di esecuzione sono formattati in blocchi con sfondo nero:

```
questo è l’output del mio programma
```

> **Infine, le note come questa aggiungono informazioni all’argomento trattato prima o dopo di esse. Possono essere approfondimenti, suggerimenti, o anche semplici rimandi ad altre parti dell’opera.**

Icone semantiche

Per aiutare visivamente ad identificare parti del testo con particolari caratteristiche semantiche, sono state utilizzate le seguenti icone.

Novità di Java 9

Best practice

Concetto complesso

Utilizzo non comune

Errore comune

Molto importante

A chi si rivolge

Manuale di Java 9 è stato strutturato per soddisfare le aspettative di:

- Aspiranti programmatori che vogliono entrare nel mondo del lavoro: nulla è dato per scontato, è possibile imparare a programmare anche partendo da zero, ed entrare nel mondo del lavoro dalla porta principale.
- Studenti universitari: le precedenti versioni di questa opera sono state adottate come libro di testo per molti corsi universitari nelle maggiori università italiane.
- Programmatori Java esperti che devono aggiornarsi alla versione 9: ricordiamo che la modularizzazione della piattaforma ha rivoluzionato completamente il modo in cui concepiamo le applicazioni Java.

Materiale on line

Per questa edizione tantissimo materiale è stato spostato on line. Stiamo parlando di un numero di pagine che quasi eguaglia le pagine del libro cartaceo, e che avrebbe potuto tranquillamente dar luogo ad un secondo volume! In particolare tra le ventuno appendici si trovano anche alcuni capitoli che originariamente dovevano far parte del libro (interfacce grafiche, Java FX, interazione con database, etc.), che arricchiscono e completano l’opera. Inoltre, gli esercizi sono raddoppiati rispetto all’edizione precedente, e sono disponibili on line per non togliere spazio alla teoria all’interno del

libro. Potete scaricare tutti gli esercizi e le appendici in un apposito spazio disponibile on line raggiungibile agli indirizzi http://www.hoeplieditore.it/8302-2 e http://www.claudiodesio.com/java9.html.

Codice sorgente

Lavorando con gli esempi di codice presenti in questo libro, si può decidere di scrivere tutto il codice a mano oppure di usare i file sorgenti che accompagnano il libro. Tutto il codice sorgente utilizzato in questo libro è disponibile per il download agli indirizzi http://www.hoeplieditore.it/8302-2 e http://www.claudiodesio.com/java9.html.

Errata

È stato compiuto ogni sforzo possibile per evitare errori nel testo e nel codice. Tuttavia, nessuno è perfetto, e gli sbagli capitano. Se si dovesse trovare un errore ortografico o una parte di codice non funzionante, la casa editrice sarebbe grata per la segnalazione che sarà messa in errata corrige. Altri lettori eviteranno così ore di frustrazione e allo stesso tempo si contribuisce a garantire un livello di qualità dell'informazione sempre maggiore. La segnalazione si può fare direttamente all'autore scrivendo a claudio@claudiodesio.com, oppure all'indirizzo hoepli@hoepli.it. Per trovare l'errata in italiano sul web, si faccia riferimento agli indirizzi http://www.hoeplieditore.it/8302-2 e http://www.claudiodesio.com/java9.html.

Autore

Dal 1999 lavoro come consulente IT free lance. Oggi sono specialista in formazione, technical writing, sviluppo, analisi, progettazione, tecnologie Java, architettura e metodologie object oriented. Sono autore di diversi articoli tecnici e di libri di grande successo: "Manuale di Java 6", "Manuale di Java 7" e "Manuale di Java 8", tutti editi da Hoepli. Ho collaborato con diverse Università, Enti Ministeriali e aziende tra cui Sun Microsystems (ora Oracle) creatrice di Java, come trainer e mentor. È possibile contattarmi

tramite i seguenti mezzi:

- Internet: http://www.claudiodesio.com
- E-mail: claudio@claudiodesio.com
- Facebook: http://www.facebook.com/claudiodesiocesari
- Twitter: http://twitter.com/cdesio
- LinkedIn: http://www.linkedin.com/in/claudiodesio
- Google +: http://plus.google.com/+ClaudioDeSioCesari

Ringraziamenti

Il mio ringraziamento è per tutte le persone che mi hanno sostenuto in questo lavoro, e a tutti coloro che in qualche modo hanno contribuito alla sua realizzazione (supportandomi o sopportandomi!).

Un grazie particolare va a Emanuele Giuliani che ha curato anche quest'opera con grande pazienza e professionalità.

Inoltre ringrazio (ma mai abbastanza) i miei genitori, per tutto quello che hanno fatto per me e che continuano a fare, e per il loro amore incondizionato.

Infine ringrazio la mia famiglia, ovvero la mia ragione di vita: Rosalia, Andrea e Simone, a voi è dedicato anche questo libro. Siete tutto per me.

Claudio De Sio Cesari

A Rosalia, Andrea e Simone...

Parte I

Le basi del linguaggio

La parte I è stata concepita per permettere un approccio a Java “non traumatico”. Anche il lettore senza esperienze di programmazione e a digiuno di fondamentali concetti di informatica, può iniziare a programmare da zero! I concetti vengono introdotti in maniera graduale seguendo un metodo didattico perfezionato in vent’anni di erogazione di corsi di formazione su Java. Nel primo capitolo si viene introdotti all’ambiente di sviluppo per scrivere subito un primo semplice programma. Nel secondo è presente un’esauriente copertura dei concetti fondamentali del linguaggio quali classi, oggetti, metodi e così via. Il terzo capitolo è dedicato essenzialmente ai tipi di dati di Java. Nel quarto vengono presentati tutti gli strumenti per gestire il flusso di controllo offerti dal linguaggio. Infine il quinto capitolo, oltre a JShell (una delle grandi novità di Java 9), introduce il lettore al ciclo di sviluppo del software, agli IDE, e soprattutto definisce un metodo per sviluppare un programma da zero passo dopo passo. Al termine dello studio di questa prima parte, il lettore dovrebbe possedere tutte le carte in regola per iniziare a programmare in Java.

Lo studio di questa sezione risulterà ugualmente utile al programmatore esperto che ha bisogno solo di aggiornare le proprie conoscenze alla versione 9 di Java; queste nuove informazioni saltano subito all’occhio grazie all’icona “Novità di Java 9”.

1

Introduzione a Java

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:
- ✔ Conoscere alcuni concetti informatici di base (unità 1.1).
- ✔ Saper definire il linguaggio di programmazione Java e le sue caratteristiche (unità 1.2).
- ✔ Interagire con l'ambiente di sviluppo: il Java Development Kit (unità 1.3, 1.4, 1.5).
- ✔ Saper digitare, compilare e mandare in esecuzione una semplice applicazione (unità 1.3, 1.4, 1.5).
- ✔ Scegliere un ambiente di sviluppo (unità 1.6).

L'obiettivo primario di questo capitolo è quello di permettere al lettore di ottenere subito dei primi risultati concreti. Al termine del capitolo avremo creato ed eseguito un primo programma Java. Si tratta ovviamente di un programma molto semplice, ma questo ci permetterà di prendere subito confidenza con la programmazione Java. Per raggiungere lo scopo non daremo nulla per scontato: definiremo il linguaggio e la tecnologia Java, descriveremo l'ambiente di sviluppo, mostreremo un processo che passo dopo passo ci permetterà di eseguire la nostra prima applicazione, impareremo a comprendere gli eventuali messaggi di errore e a risolvere i primi problemi di programmazione.

È del tutto probabile che senza basi di programmazione si incontrino delle difficoltà nel capire alcuni concetti, ma non bisogna scoraggiarsi. In questo primo capitolo c'è tutto il necessario per iniziare da zero. Bisogna però essere sempre concentrati e non è consigliabile saltare frasi o addirittura paragrafi. Se qualcosa non risultasse chiaro è consigliata una rilettura.

1.1 Prerequisiti: concetti informatici di base

Per poter programmare bisogna conoscere alcuni concetti base di informatica. In questa nuova edizione è stato deciso di introdurre il paragrafo che state leggendo per permettere a tutti di approcciare alla programmazione. In questo modo anche chi è completamente a digiuno non solo di programmazione, ma anche dei concetti che sono alla base del funzionamento di un computer, potrà iniziare a studiare senza dover ricorrere continuamente a Google. Per tale ragione anche durante la lettura dei successivi capitoli troverete digressioni sui concetti base. Avere un vocabolario comune è il primo passo per poter comprendere a fondo questo libro. Cercheremo comunque di limitare le spiegazioni solo ai concetti davvero necessari, ed ovviamente utilizzeremo la sintesi per non sotterrare la voglia di apprendere sotto il peso di troppe informazioni superflue.

Iniziamo da zero dando alcune definizioni di cui faremo largo uso in questo libro. Per poter programmare in Java, avete bisogno di un computer, un dispositivo elettronico che aggrega una serie di componenti fisici a cui ci si riferisce solitamente con il termine **hardware.** Esempi di hardware sono tutti i componenti di interfaccia con cui l'utente interagisce: il mouse, la tastiera, il monitor, la stampante, etc. In un computer però, trovano posto anche tanti altri componenti hardware meno "visibili" come il processore, la scheda madre e la memoria RAM.

Un computer però ha bisogno di **istruzioni** per poter funzionare ed eseguire operazioni. Tali istruzioni sono definite dai **programmi**, a cui ci si riferisce in generale con il termine **software.** Quindi un computer è definito dalla sua parte fisica (hardware) e dai programmi che esegue (software).

Oggigiorno esistono tantissime tipologie di dispositivi che possono essere definiti computer. In questo libro diamo per assunto che il lettore utilizzi un computer fisso (**desktop**) o portatile (**laptop**), e non un tablet, uno smartphone o altri dispositivi.

1.1.1 Hardware

In questo contesto distingueremo tre tipologie di hardware. La prima tipologia si occupa di raccogliere dati di **input** ("in ingresso"). Appartengono a questa prima tipologia dispositivi come il **mouse** o la **tastiera**, che hanno il compito di accettare input dall'utente del computer. La seconda tipologia ha come compito quello di restituire dati in **output** ("in uscita"). Esempi di componenti che appartengono a questa seconda tipologia sono il **monitor** e la **stampante.** Queste due tipologie sono da considerare "hardware d'interfaccia" ovvero quelle con cui l'utente interagisce. Per tale ragione dovreste già avere una conoscenza basilare di questi dispositivi.

La tipologia di hardware che più ci interessa approfondire invece, è quella con cui non interagiamo direttamente: quella "interna alla scatola". Prima di parlare di programmi infatti, bisogna avere almeno un'idea di come un computer memorizza i dati, ovvero come funzionano per esempio la memoria e il processore.

Il **processore** o **CPU** (acronimo di "central processing unit" ovvero "unità di elaborazione centrale") è il componente hardware che esegue le istruzioni di un programma. Il suo principale compito è quello di eseguire operazioni aritmetiche elementari, e gestire gli spostamenti dei dati in memoria. Può sembrare strano, ma è il processore che rende possibile tutto quello che riusciamo a fare con un computer.

La **memoria** conserva i dati che vengono utilizzati dai programmi e bisogna distinguere due tipologie: una memoria principale ed una ausiliaria. La **memoria principale** solitamente è detta **memoria RAM** (acronimo per "Random Access Memory" ovvero "memoria ad acceso casuale") ed è una memoria che contiene i programmi in esecuzione con i relativi dati. Questa memoria è temporanea (si dice anche "volatile") e viene cancellata quando il computer viene spento. La memoria ausiliare invece conserva i dati anche a computer spento. La **memoria ausiliare** è quindi solitamente molto più estesa di quella principale, ma anche meno performante dal punto di vista

della velocità di immagazzinamento dati, ed è presente in un computer sotto forma di hard disk (in italiano “disco rigido”) e altri dispositivi come memorie a stato solido, penne USB etc.

1.1.2 Software

Per utilizzare un computer c’è bisogno di programmi che contengano istruzioni da far eseguire all’hardware sottostante. In particolare un computer esegue almeno un programma principale che lo caratterizza e spesso viene identificato con il computer stesso: il **sistema operativo.** Oggigiorno i più famosi sistemi operativi sono i sistemi Windows della Microsoft, i Mac OS della Apple, e i sistemi Linux e Unix. Quando accendiamo un computer viene eseguito il suo sistema operativo che si occupa di gestire il funzionamento del computer nel suo insieme. Esso ci mette a disposizione tra l’altro la possibilità di installare ed eseguire altri programmi. Per esempio il sistema operativo Microsoft Windows 10 consente di installare ed eseguire programmi come Microsoft Word, un browser come Mozilla Firefox o Google Chrome, un IDE di sviluppo Java come Netbeans o Eclipse, un lettore multimediale come VLC, oppure un programma scritto in Java da noi.

Esistono migliaia di tipologie di programmi molto diversi tra di loro. Per esempio un browser è un programma che permette di navigare in internet. Un IDE invece è un ambiente di programmazione che integra diversi strumenti (“IDE” è l’acronimo per “Integrated Development Environment” ovvero “ambiente di sviluppo integrato”). Un lettore multimediale permette invece di visualizzare video o ascoltare audio. Quindi ogni programma esegue istruzioni che molto spesso si basano su dati di input, eseguono determinate attività e producono un risultato come output. Per esempio un browser può prendere in input un indirizzo internet inserito dall’utente mediante la tastiera. Può eseguire un processo che comprende l’individuazione della risorsa richiesta on line, il recupero del codice che costituisce la pagina web, l’interpretazione dei linguaggi e delle tecnologie che vengono utilizzate nel codice della pagina. Può infine produrre in output la formattazione della pagina così come la vede l’utente.

1.1.3 Linguaggi di programmazione

Un **linguaggio di programmazione** è un mezzo che permette di comunicare delle istruzioni al computer in modo che esso le esegua nel contesto di un programma. Si tratta appunto di un linguaggio con un suo vocabolario e le sue regole. Il vocabolario è quasi sempre piuttosto scarno ma le regole sono solitamente tante.

> **L'insieme delle sue regole è detto talvolta grammatica del linguaggio.**

Esistono tantissimi linguaggi di programmazione e la storia della loro nascita e della loro storia è affascinante. Linguaggi come Assembly, Fortran, Basic, COBOL, Pascal, C, C++, LISP, Prolog, SmallTalk, Visual Basic, Javascript e Java, sono solo alcuni dei linguaggi che hanno fatto la storia della programmazione (la lista sarebbe lunghissima). La domanda sorge spontanea: un computer riesce a comprendere tutti questi linguaggi? Ovviamente no! L'hardware di un computer è in grado di comprendere un unico linguaggio, detto **linguaggio macchina.** Il linguaggio macchina ha un vocabolario definito dal **sistema numerico binario** e quindi formato solamente da due simboli: `0` e `1`. Componendo in sequenza questi simboli si ottengono delle "parole", e con tali parole è possibile creare delle istruzioni da far eseguire al processore.

> **Esistendo diverse tipologie di processori è scorretto tecnicamente parlare dell'esistenza di un unico linguaggio macchina, perché in realtà ne esistono tanti. Ogni processore sa interpretare un particolare linguaggio macchina.**

Allora come possiamo creare un programma per computer di cui non si conosce a priori il processore e quindi il linguaggio macchina? E come fa un computer a capire un programma scritto in un qualsiasi linguaggio di programmazione quando è capace di comprendere solo il linguaggio macchina? Sembra che manchi proprio un "traduttore", e infatti è così.

I linguaggi moderni come Java, sono detti **linguaggi di alto livello**, ovvero linguaggi che sono molto vicini al linguaggio che noi utilizziamo ogni giorno per comunicare, e di conseguenza molto distanti dal linguaggio macchina (che ricordiamo può usare solo i simboli `0` e `1` per formulare le sue istruzioni). Invece i linguaggi come l'Assembly che sono invece molto simili al linguaggio macchina sono detti **linguaggi di basso livello.** Ma sia i programmi scritti con Assembly che quelli scritti con Java richiedono una traduzione in linguaggio macchina per essere compresi dal processore. Questa traduzione viene fatta da un software che fa parte dell'ambiente di programmazione del linguaggio, chiamato **compilatore.** Un compilatore quindi è un software che traduce le istruzioni scritte con un linguaggio di programmazione in istruzioni scritte in linguaggio macchina.

II processo di sviluppo di un programma comprenderà i seguenti passi:

1. il programmatore utilizzando un linguaggio di programmazione scrive un programma. Rispettando le regole del linguaggio salva le istruzioni all'interno di uno o più file testuali. Si dice che tali file contengono il **codice sorgente** del programma, ed i file testuali stessi vengono detti **file sorgente.**
2. il programmatore utilizza un compilatore a cui darà in input i file sorgente. Il compilatore, dopo aver verificato la correttezza del codice sorgente, tradurrà le istruzioni dal codice sorgente in istruzioni scritte in linguaggio macchina. Per fare questo il compilatore creerà a partire da file di testo i cosiddetti **file binari.** I file binari sono direttamente eseguibili dal processore ogni volta che verranno eseguiti, e rappresentano il programma stesso. I programmi pronti all'uso vengono anche detti **file eseguibili**, o **file oggetto.**

I linguaggi che godono del supporto di un compilatore vengono detti **linguaggi compilati.**

Esistono però anche linguaggi che non usano un compilatore per tradurre il codice sorgente in linguaggio macchina ma un software che viene chiamato **interprete.** A differenza di un compilatore, un interprete è uno strumento che riesce ad eseguire il programma, traducendo al volo il codice sorgente in codice oggetto, istruzione dopo istruzione. Un linguaggio che utilizza un interprete invece di un compilatore viene detto **linguaggio interpretato.** Se

da un lato un interprete può rappresentare una soluzione più flessibile nella dinamica di sviluppo facendoci alternare la scrittura del codice e la sua esecuzione senza doverlo compilare, soffre di uno svantaggio evidente. Infatti la traduzione del codice sorgente in codice macchina è compresa nella fase di esecuzione del programma che inevitabilmente è più lenta rispetto all'esecuzione di un programma che è stato compilato. Inoltre quest'alternanza tra traduzione ed esecuzione avviene ogni volta che il programma viene interpretato.

Ogni linguaggio di programmazione richiede comunque un compilatore o un interprete diverso, a seconda dell'hardware dove il programma deve essere eseguito. In molti linguaggi questo può provocare comportamenti diversi dei programmi lanciati su computer diversi e generati a partire dallo stesso sorgente. Questo problema però, è stato brillantemente risolto da Java come vedremo presto.

1.2 Introduzione a Java

Dopo la breve parentesi iniziale che ha introdotto alcuni concetti basilari di informatica, ora è il momento di una breve introduzione su Java. Non ci dilungheremo però narrandovi delle eroiche gesta dei creatori di Java, di tutte le leggende e le previsioni per il futuro. Diversi argomenti infatti non troveranno posto tra i paragrafi di questo libro perché trasformati in apposite appendici da scaricare on line. Tra queste appendici troverete anche quella relativa alla creazione e alla storia di Java: l'appendice A. Se siete curiosi di sapere perché il linguaggio si chiama Java, chi sono i creatori, quali sono le condizioni che hanno determinato la nascita e il successo di Java etc., dovete leggere l'appendice A. In queste pagine invece cercheremo di puntare diritto al nostro obiettivo, per non far annoiare chi non vede l'ora di mettere mano al codice.

Per contenere il numero di pagine e di conseguenza il costo di questo libro, tantissimo materiale è stato spostato on line! È possibile scaricare decine di appendici, oltre 300 pagine di esercizi, tutto il codice presentato nel libro e l'errata corrige visitando l'indirizzo

http://www.claudiodesio.com/java9.html.

Da qui sarete reindirizzati su http://www.hoepli.it, nella pagina dedicata a questo libro, dove troverete i link per scaricare tutto il materiale.

1.2.1 Cos'è Java

Con il termine "Java", solitamente ci si riferisce:

1. al linguaggio di programmazione più usato del pianeta;
2. ad una tecnologia che include diverse "sotto-tecnologie" che si sono affermate in diversi ambiti di utilizzo del software. Oggigiorno la tecnologia Java è la più utilizzata su applicazioni enterprise, e in tutto il mondo esistono miliardi di congegni elettronici che utilizzano tecnologia Java: smartphone, SCADA, telefoni cellulari, satelliti, decoder, smart card, robot che passeggiano su Marte, etc.

In questo libro ci concentreremo sul linguaggio di programmazione Java che è il punto di partenza per poter accedere a tutte le tecnologie Java. Accenneremo e introdurremo alcune di esse, ma quel tanto che basta per suscitare curiosità nel lettore, che poi se vorrà dovrà approfondire lo studio di tali tecnologie con altre risorse didattiche.

1.2.2 Caratteristiche del linguaggio Java

Il linguaggio fu presentato nel 1995 da Sun Microsystems, storica e gloriosa società americana che dal 2010 è stata assorbita da Oracle.

Le caratteristiche principali di Java sono:

- **sintassi:** è simile a quella di altri linguaggi di programmazione storici come il C e il C++. Altri linguaggi con sintassi molto simile sono linguaggi che sono nati dopo Java come C# e JavaScript. Questo significa che se il lettore ha già conoscenze di uno di questi linguaggi, avrà un vantaggio evidente in questa prima parte dello studio. Tuttavia è una sintassi considerata tra le più chiare tra i linguaggi di programmazione, caratterizzata da un'alta leggibilità (a volte sembra

di scrivere in linguaggio naturale).

- **Robustezza:** un linguaggio è più robusto di un altro quando ha maggiori capacità di prevenire gli errori di programmazione, e Java è uno dei linguaggi più robusti in circolazione. Definisce una gestione delle eccezioni chiara e funzionale, e possiede un meccanismo automatico della gestione della memoria (Garbage Collection) che esonera il programmatore dall'obbligo di dover deallocare memoria quando ce n'è bisogno (punto tra i più delicati nella programmazione). Inoltre il compilatore Java, è molto "severo". Il programmatore è infatti costretto a risolvere tutte le situazioni poco chiare, garantendo al programma maggiori chance di corretto funzionamento. La logica è: "è molto meglio ottenere un errore in compilazione che in esecuzione".
- **Indipendenza dalla piattaforma:** grazie al concetto di Java Virtual Machine, ogni applicazione, una volta compilata, potrà essere eseguita su di una qualsiasi piattaforma (per esempio un PC con sistema operativo Windows o una workstation Unix). Questa è sicuramente la caratteristica più importante di Java. Infatti, nel caso in cui si debba implementare un programma destinato a girare su diverse piattaforme, non ci sarà la necessità di dover creare versioni specifiche differenti. Si tratta ovviamente di un vantaggio enorme.
- **Java Virtual Machine:** ciò che rende di fatto possibile l'indipendenza dalla piattaforma, è la Java Virtual Machine (da ora in poi **JVM).** Si tratta di un software capace di svolgere il ruolo di interprete. Java in effetti può essere considerato un linguaggio sia compilato che interpretato. Infatti, dopo aver scritto il nostro codice sorgente con Java, dovremo comunque prima utilizzare il compilatore java per compilarlo. Otterremo però, non direttamente un file eseguibile contenente istruzioni in linguaggio macchina, ma un file che contiene la traduzione del nostro listato in un linguaggio molto vicino al linguaggio macchina detto **bytecode.** Una volta ottenuto questo file, il programmatore potrà passarlo in input alla JVM che interpreterà il bytecode ed il nostro programma verrà finalmente eseguito. Quindi, se una piattaforma qualsiasi possiede una Java Virtual Machine, ciò

sarà sufficiente per renderla esecutrice di bytecode. È il caso del sistema operativo mobile più diffuso (Android di Google) dove le applicazioni native sono scritte in Java. Ecco perché esistono così tanti modelli diversi di smartphone e tablet (ma anche di fornelli, lavatrici, frigoriferi, robot etc.) che usano il sistema operativo Android, il segreto è la virtual machine. Dopo aver tradotto in bytecode il nostro codice sorgente, il nostro programma è potenzialmente eseguibile su tantissime tipologie di dispositivi. La Java Virtual Machine è detta "macchina virtuale" perché questo software è stato implementato per simulare un hardware. Si potrebbe affermare che "il linguaggio macchina sta ad un computer come il bytecode sta ad una Java Virtual Machine". Oltre che permettere l'indipendenza dalla piattaforma, la JVM garantisce a Java tanti altri vantaggi. Per esempio quello di essere un linguaggio multi-threaded (caratteristica di solito dei sistemi operativi) ovvero capace di mandare in esecuzione più processi in maniera parallela. Inoltre, garantisce dei meccanismi di sicurezza molto potenti, la "supervisione" del codice da parte del Garbage Collector per la gestione automatica della memoria, e tanto altro.

- **Orientato agli oggetti:** la **programmazione Object Oriented** (letteralmente "orientata agli oggetti") è uno stile di programmazione fondamentale che ci permette di programmare in maniera più "naturale". Parleremo ampiamente di questo argomento a partire dal sesto capitolo. Java ci fornisce infatti alcuni strumenti che praticamente ci "obbligano" a programmare ad oggetti. I paradigmi fondamentali della programmazione ad oggetti (ereditarietà, incapsulamento, polimorfismo) sono più facilmente apprezzabili e comprensibili. Java è più chiaro e schematico di qualsiasi altro linguaggio orientato agli oggetti. Sicuramente, chi impara Java, potrà in un secondo momento accedere in modo più naturale alla conoscenza di altri linguaggi orientati agli oggetti, giacché avrà di certo un modo di pensare alla programmazione già orientato agli oggetti. Tuttavia con il passare del tempo il linguaggio si sta evolvendo, e si è aperto anche ad altri paradigmi di programmazione.
- **Facilità di sviluppo:** prima della pubblicazione della versione 5, Java

era pubblicizzato come un linguaggio semplice da imparare. Dalla pubblicazione della release 5 in poi il termine *simplicity* (letteralmente "semplicità") è stato sostituito con *ease of development* (in italiano "facilità di sviluppo"). Infatti, con l'introduzione di rivoluzionarie caratteristiche come i generics o le annotazioni, ci si è incanalati in una strada delineata chiaramente: quella di far evolvere il linguaggio per non perdere il treno della modernità, anche a scapito della semplicità. Il linguaggio oggi è diventato più difficile da imparare, ma i programmi dovrebbero essere più semplici da scrivere. Insomma, imparare Java non sarà una passeggiata, ma una volta comprese le logiche fondamentali avremo tra le mani uno strumento potentissimo.

- **Libreria e standardizzazione:** Java possiede un'enorme libreria di classi standard, ottimamente documentate. Ciò rende Java un linguaggio di altissimo livello e permette anche ai neofiti di creare applicazioni complesse in un tempo abbastanza breve. Per esempio, è relativamente semplice creare interfacce grafiche, collegamenti a database e connessioni di rete. E questo indipendentemente dal computer su cui si sviluppa. Inoltre, grazie alle specifiche di Oracle, non esisteranno per lo sviluppatore problemi di standardizzazione come compilatori che compilano in modo differente.

- **Aperto:** utilizzare Java non significa muoversi in un ambiente chiuso dove tutto è standard. Dalla scelta del tool di sviluppo (Eclipse, Netbeans, etc.) all'interazione con altri linguaggi, librerie esterne e tecnologie (SQL, XML, framework open source, etc.) non si finisce mai di imparare! Il codice di Java è inoltre Open Source, il che significa che è possibile leggerne il codice sorgente. Dalla sua nascita Java è sempre stato considerato un linguaggio "aperto", e l'interazione con gli sviluppatori è stato sempre un punto di forza della sua evoluzione. Il sito di riferimento oggi è http://openjdk.java.net/ ed è possibile anche attivamente contribuire allo sviluppo di Java con test, proposte e bug fixing. Le nuove caratteristiche delle ultime versioni sono tutte state proposte da sviluppatori che hanno dato il loro contributo con le idee. Java è fatto dagli sviluppatori per gli sviluppatori.

Esistono tante altre caratteristiche che potrebbero essere menzionate, ma queste sono quelle che più dovrebbero interessare ai lettori di questo libro.

1.3 Ambiente di sviluppo

> **In questo libro daremo per scontato che il lettore utilizzi un computer su cui sia installato un sistema operativo Windows 7 o superiore (consigliato Windows 10), ma è ovviamente possibile programmare anche su altre piattaforme come Linux o Mac.**

Per scrivere un programma Java abbiamo bisogno di due software:

1. **un programma che ci permetta di scrivere il codice Java.** Per iniziare può andar bene un semplice editor di testo, come Blocco Note (Notepad) di Windows. Sconsigliamo WordPad, Word o qualsiasi altro editor che gestisca stili, formattazioni, etc.
2. **Il Java Development Kit versione Standard Edition** (da ora in poi **JDK).** Si tratta dell'ambiente di sviluppo ufficiale di Java. All'interno troviamo una suite di software tra i quali un compilatore e la Java Virtual Machine di cui abbiamo accennato nel paragrafo precedente. Il JDK è scaricabile gratuitamente a questo indirizzo: http://www.oracle.com/technetwork/java/javase/downloads/index.html con le relative note d'installazione e documentazione. Nell'appendice B (da scaricare all'indirizzo http://www.claudiodesio.com/java9.html) è descritto il processo da seguire per procurarsi ed installare correttamente l'ambiente di sviluppo su di un sistema operativo Windows. Alternativamente potete utilizzare anche la video-guida presente sul canale YouTube di questo manuale all'indirizzo http://www.claudiodesio.com/yt/jdk.

> **Altri ambienti di sviluppo alternativi saranno discussi alla fine del capitolo.**

1.3.1 Struttura del JDK

Una volta installato correttamente il JDK, questo dovrebbe trovarsi nella cartella `C:\Program Files\Java`. Nella versione 9, la struttura delle cartelle che il JDK contiene è stata rivoluzionata. Rispetto alla versione precedente alcune cartelle sono state rimosse, altre invece sono state aggiunte. Segue una breve spiegazione delle attuali cartelle:

- **bin:** contiene tutti i file eseguibili del JDK, ovvero `javac`, `java`, `jar`, `javadoc`, che ci permetteranno di compilare, eseguire, creare archivi di file, etc.
- **conf:** contiene dei file di configurazione che usa il JDK.
- **include** e **lib:** contengono librerie scritte in C e in Java che sono utilizzate dal JDK.

> **Con il termine "libreria" intendiamo un software che offre la possibilità ad altri software (nel nostro caso il JDK) di utilizzare le funzionalità che definisce.**

- **legal:** contiene i file delle licenze del JDK.
- **jmods:** contiene la libreria standard di Java riorganizzata in moduli. La più importante delle novità di questa versione riguarda proprio l'introduzione di questo nuovo modo di organizzare i nostri programmi: la modularizzazione. Vedremo solo alla fine del libro come utilizzare i moduli in un capitolo dedicato.

Inoltre accanto alla cartella JDK è stata installata anche un'altra cartella: la cartella del **Java Runtime Environment (JRE).** Affinché un'applicazione Java risulti eseguibile su di una macchina, basta installare solo il JRE (è scaricabile anche indipendentemente dal JDK da http://www.java.com). Esso rappresenta essenzialmente la JVM con il supporto per le librerie della versione corrente di Java. Il JRE viene installato in questa cartella

automaticamente quando viene installato il JDK.

Riassumendo, la differenza tra JVM, JRE e JDK:

La JVM (Java Virtual Machine, in italiano "macchina virtuale di Java") è la macchina virtuale: il software che simula un hardware capace di interpretare ed eseguire il bytecode contenuto in un file Java compilato.

Il JRE (Java Runtime Environment, in italiano "ambiente di esecuzione di Java") consiste in un software che mette a disposizione l'ambiente per utilizzare la JVM (e infatti la contiene). Per potere eseguire codice Java bisogna scaricare il JRE (non è possibile scaricare solo la JVM).

Il JDK (Java Development Kit, in italiano "Kit di sviluppo Java") è il software che serve per sviluppare in Java. Contiene una suite di strumenti per sviluppare. Essenzialmente, il JDK serve per sviluppare in Java, il JRE per eseguire programmi Java. A noi serve il JDK.

1.3.2 Guida per sviluppare passo dopo passo

Dopo aver installato correttamente il JDK e impostato opportunamente le eventuali variabili di ambiente (cfr. appendice B), saremo pronti per scrivere la nostra prima applicazione (prossimo paragrafo).

Se state utilizzando un sistema Windows, una raccomandazione è quella di abilitare la visualizzazione del suffisso di tutti i file, visto che i programmatori devono sempre distinguere tipologie di file diversi. Purtroppo sui sistemi Windows l'impostazione di default prevede che per i file conosciuti non sia visibile il suffisso. Per risolvere la situazione bisogna: aprire il Pannello di controllo **di**

Windows, … selezionare la voce Opzioni Esplora file **(sui vecchi sistemi Windows selezionate la voce Esplora Cartella), selezionare il tab Visualizzazione e deselezionare il checkbox Nascondi le estensioni per i tipi di file conosciuti. Dopo aver fatto questo vedremo i suffissi di ogni singolo file sul nostro hard disk.**

In generale ogni volta che programmiamo dovremo eseguire i seguenti passi:

1. **scrittura del codice:** scriveremo il codice sorgente della nostra applicazione utilizzando un editor. Come già asserito precedentemente consigliamo, solo per questa prova iniziale, Blocco Note di Windows.
2. **Salvataggio:** tramite l'applicazione Blocco Note, salveremo il nostro file con suffisso `.java` nella memoria secondaria, presumibilmente su un hard disk.
3. **Apertura di una prompt dei comandi:** una volta salvato il nostro file Java, dobbiamo aprire una finestra **prompt dei comandi** (detta anche **prompt di DOS).**

Se non si trova il collegamento per aprire la prompt dei comandi, per aprirla è possibile digitare il comando cmd **nel campo di ricerca del menu start di Windows, e premere il tasto** invio (**o** enter). **Se non siete pratici del sistema operativo DOS (sistema originale che si evolse nelle prime versioni di Windows), nell'appendice C potete trovare un breve tutorial che vi permetterà di muovervi anche in questo ambiente.**

4. **Posizionamento:** da questa prompt dobbiamo spostarci (consultare l'appendice C se non si è in grado di farlo) nella cartella in cui è stato salvato il nostro file sorgente.

5. **Compilazione:** digitare il comando di compilazione:

```
javac nomeFile.java
```

   e premere il tasto invio per eseguire. Se la compilazione ha esito positivo, verrà creato un file chiamato `nomeFile.class`, nella stessa cartella dove è presente il file `nomeFile.java`. Nel file `.class`, come abbiamo già detto, ci sarà la traduzione in bytecode del file sorgente.

6. **Esecuzione:** a questo punto potremo mandare in esecuzione il programma invocando l'interpretazione della Java Virtual Machine. Basta scrivere dalla prompt di DOS il seguente comando:

```
java nomeFile
```

   (senza suffissi). Premere il tasto `invio` per confermare.

L'applicazione, a meno di errori nel codice, verrà eseguita dalla JVM. In particolare il **class loader** (letteralmente "caricatore di classi") della Java Virtual Machine farà in modo che il bytecode presente nel file `.class` sia caricato nella memoria principale (la memoria RAM) per essere eseguito. Poi il **bytecode verifier** (letteralmente "verificatore di bytecode") della JVM verificherà che il codice non violi le restrizioni di sicurezza imposte da Java e non danneggi il computer (come potrebbe fare un virus). Infine tramite la **compilazione JIT** (acronimo di **"just in time"**, che potremmo tradurre come "appena in tempo") la JVM eseguirà silenziosamente una seconda fase di compilazione parallela all'interpretazione del bytecode. Infatti, individuerà e compilerà in linguaggio macchina le parti di bytecode che vengono utilizzate più volte (chiamate **hot spot)** e non dovranno essere più interpretate perché saranno già state tradotte in codice macchina. Questo favorisce la performance del nostro programma.

Ora siamo pronti per scrivere il nostro primo programma Java.

1.4 Primo approccio al codice

Diamo subito uno sguardo alla classica applicazione `Hello World`. Si tratta del tipico primo programma che si scrive quando si impara un nuovo linguaggio di programmazione. In pratica è un semplice programma che

stampa a video la frase “Hello World!” (letteralmente “Ciao Mondo!”). In questo modo inizieremo a familiarizzare con la sintassi e con qualche concetto fondamentale come quello di classe e di metodo. Avvertiamo il lettore che inevitabilmente qualche punto rimarrà oscuro, e che quindi bisognerà dare per scontate alcuni parti di codice. Questo è l’approccio didattico che seguirà questo libro: invece di cercare di spiegare alcuni concetti prematuramente, il lettore verrà a volte rimandato a capitoli successivi per la loro piena comprensione.

Vedremo anche come compilare e come mandare in esecuzione il nostro mini programma. Come già detto, il suo compito è quello di stampare a video (nella stessa prompt dei comandi) il messaggio “Hello World!”. Segue il listato:

```
1 public class HelloWorld
2 {
3     public static void main(String args[])
4      {
5         System.out.println(“Hello World!”);
6      }
7 }
```

I numeri di riga non fanno parte dell’applicazione (non bisogna scriverli) ma ci saranno utili per la sua analisi.

Questo programma deve essere scritto utilizzando l’applicazione Blocco Note. Poi deve essere salvato utilizzando esattamente il nome della classe che dichiara (prestando attenzione anche alle lettere maiuscole e minuscole) e con suffisso `.java`. Il nome del file che conterrà il listato appena presentato dovrà essere quindi `HelloWorld.java`.

Sconsigliamo il “copia - incolla” del codice. Almeno per i primi tempi, si cerchi di scrivere tutto il codice possibile. Consigliamo al lettore di scrivere il programma con Blocco Note riga dopo riga dopo averne letto la seguente analisi. In questo modo prenderà più coscienza di quello che scrive.

1.4.1 Analisi del programma HelloWorld

Analizziamo quindi il listato precedente riga per riga.

Riga 1:

```
1 public class HelloWorld
```

Dichiarazione della classe chiamata `HelloWorld`. Come vedremo ogni applicazione Java è costituita da classi. Il concetto di classe sarà approfondito nel successivo paragrafo. Per ora dobbiamo comprendere solo che le tutte le istruzioni di programmazione operative saranno sempre contenute in classi. Dobbiamo inoltre tenere presente che programmi costituti da un'unica classe (e quindi da un solo file) sono solitamente programmi d'esempio (come quello che abbiamo appena presentato). In realtà i programmi Java sono solitamente composti da più classi, ognuna dichiarata in un proprio file. Ogni classe ha una visibilità che ne determina l'utilizzabilità da parte di eventuali altre classi. La classe `HelloWorld` è stata dichiarata accessibile in maniera *pubblica* (il grado più libero di accessibilità) mediante la dichiarazione del modificatore `public`. In java un **modificatore** caratterizza una classe, così come in italiano un aggettivo può caratterizzare un sostantivo. In questo caso si dice quindi che la classe `HelloWorld` è pubblica. Ovviamente anche questo argomento sarà affrontato in dettaglio nei prossimi capitoli.

Riga 2:

```
2  {
```

Questa parentesi graffa aperta indica l'inizio della classe `HelloWorld`, che si chiuderà alla riga 7 con una parentesi graffa chiusa. Il blocco di codice compreso da queste due parentesi definisce la classe `HelloWorld`.

Sulle tastiere italiane non esiste un tasto per stampare le parentesi graffe direttamente. Noi italiani dobbiamo utilizzare una combinazione di tasti. Di seguito i due metodi più utilizzati per scrivere le parentesi graffe:
1) tenere premuti i tasti CONTROL - MAIUSC - ALT, e poi il tasto con il simbolo della parentesi quadra aperta "[", per

ottenere la parentesi graffa aperta, oppure il tasto con il simbolo della parentesi quadra chiusa "]" per ottenere la parentesi graffa chiusa.
2) dopo aver azionato il tasto BLOC NUM del tastierino numerico che si trova nella parte destra di alcune tastiere, tenere premuto il tasto ALT, digitare "123" oppure "125" usando il tastierino numerico, poi rilasciare il tasto ALT. In questo modo si ottiene nel primo caso la parentesi graffa aperta e nel secondo la parentesi graffa chiusa.

Riga 3:

```
3       public static void main(String args[])
```

Questa riga è bene memorizzarla da subito, poiché essa deve essere definita in ogni applicazione Java. Trattasi della dichiarazione del metodo `main()`. In Java, il termine "metodo" è sinonimo di "azione" (i metodi saranno trattati in dettaglio nel prossimo capitolo). In particolare il metodo `main()` definisce il punto di partenza dell'esecuzione di ogni programma. La prima istruzione che verrà quindi eseguita in fase di esecuzione dal programma sarà quella che la JVM troverà subito dopo l'apertura del blocco di codice che definisce questo metodo. Oltre alla parola `main` la riga 3 contiene altre parole di cui studieremo in dettaglio il significato nei prossimi capitoli. Purtroppo, come già anticipato, quando si inizia a studiare un linguaggio ad oggetti come Java, è impossibile toccare un argomento senza toccarne tanti altri. Per adesso il lettore si dovrà accontentare della seguente tabella:

Termine	Spiegazione
`public`	Modificatore del metodo. I modificatori sono utilizzati in Java come nel linguaggio umano sono utilizzati gli aggettivi. Se si antepone un modificatore alla dichiarazione di un elemento Java (un metodo, una classe, una variabile, etc.) questo cambierà in qualche modo (a seconda del significato del modificatore) le sue proprietà. In questo caso, trattasi di uno specificatore d'accesso che rende di fatto il metodo accessibile anche al di fuori della classe in cui è stato definito.
	Modificatore del metodo. La definizione di `static` è abbastanza complessa,

`static`	per ora ci si accontenti di sapere che è essenziale per la definizione del metodo `main()`.
`void`	È il **tipo di ritorno** del metodo, ovvero la tipologia di dato che questo metodo deve restituire come output. In inglese "void" si traduce con "vuoto", e questo implica che questo metodo non restituisce nessun tipo di valore. Il metodo `main()` non deve mai avere un tipo di ritorno diverso da `void`.
`main`	Trattasi del **nome** del metodo, detto anche **identificatore** (in inglese *identifier)* del metodo.
(`String args[]`)	Alla destra dell'identificatore di un metodo, si definisce sempre una coppia di parentesi tonde che racchiude opzionalmente **una lista di parametri** (detti anche **argomenti** del metodo) che rappresentano l'input del metodo. Il metodo `main()` vuole sempre come parametro un array di `String` (agli array troveremo dedicata un'intera unità didattica nel terzo capitolo). Notare che `args` è l'identificatore (nome) dell'array ed è l'unica parola che può variare nella definizione del metodo `main()`, anche se per convenzione si utilizza sempre la parola `args`.

Riga 4:

```
4 {
```

Questa parentesi graffa indica l'inizio del metodo `main()` che si chiuderà alla riga 6 con una parentesi graffa chiusa. Il blocco di codice compreso tra queste due parentesi definisce il metodo.

Riga 5:

```
5         System.out.println("Hello World!");
```

Questo comando stamperà a video la stringa "Hello World!". Anche in questo caso, giacché dovremmo introdurre argomenti per i quali il lettore non è ancora maturo, preferiamo rimandare la spiegazione dettagliata di questo comando ai prossimi capitoli. Per ora basterà sapere che stiamo invocando un metodo che si chiama `println()` appartenente alla libreria standard di Java, passandogli come parametro la stringa che dovrà essere stampata. Questa metodo prende in input il parametro "Hello World", e lo stampa a video.

Riga 6:

```
6     }
```

Questa parentesi graffa chiusa chiude l'ultima che è stata aperta, ovvero chiude il blocco di codice che definisce il metodo `main()`.

Riga 7:

```
7 }
```

Questa parentesi graffa invece chiude il blocco di codice che definisce la classe `HelloWorld`. Notare che le parentesi erano state indentate in maniera non casuale.

1.4.2 Compilazione ed esecuzione del programma HelloWorld

Una volta riscritto il listato usando Blocco Note, dobbiamo salvare il nostro file in una cartella di lavoro, chiamata ad esempio `ManualeDiJava9`, e salvarla nella cartella radice `C:\`. Potete chiamare questa cartella di lavoro come volete e salvarla dove volete. Il consiglio è quello di avere un nome che non contenga spazi al suo interno per non introdurre inutili complicazioni. L'ideale è creare la cartella `C:\ManualeDiJava9`.

Il lettore presti attenzione al momento del salvataggio. Essendo Blocco Note un editor di testo generico, e non un editor per Java, questo tende a salvare i file con il suffisso ".txt". Quindi se proveremo a salvare il file con il nome `HelloWorld.java`, Blocco Note salverà il file `HelloWorld.java.txt`, e questo non è quello che vogliamo. Per evitare questo comportamento bisogna includere il nome del file (che ricordiamo deve essere obbligatoriamente `HelloWorld.java`) tra virgolette in questo modo "`HelloWorld.java`". Alternativamente è possibile selezionare sulla schermata di salvataggio il valore `Tutti i file (*)` nel campo `Salva come`, che si trova proprio sotto il campo di testo dove bisogna scrivere il nome del file.

A questo punto possiamo iniziare ad aprire una prompt dei comandi e spostarci all'interno della nostra cartella di lavoro. Dopo essersi accertati che il file `HelloWorld.java` sia presente nella cartella in cui ci siamo spostati (usare il comando `dir`), possiamo passare alla fase di compilazione. Se eseguiamo il comando:

```
javac HelloWorld.java
```

passiamo il nostro file sorgente in input al compilatore che ci mette a disposizione il JDK. Se al termine della compilazione non ci viene fornito nessun messaggio d'errore, vuol dire che la compilazione ha avuto successo. A questo punto possiamo notare che nella nostra cartella di lavoro è stato creato un file di nome `HelloWorld.class`. Questo è appunto il file sorgente tradotto in bytecode, pronto per essere interpretato dalla JVM. Se quindi eseguiamo il comando:

```
java HelloWorld
```

il nostro programma, se non sono sollevate eccezioni dalla JVM, sarà mandato in esecuzione, stampando il tanto sospirato messaggio:

```
Hello World!
```

1.4.3 La nostra interazione con il computer

Ecco i passi che abbiamo fatto per interagire con il computer:

1. Abbiamo chiesto al sistema operativo di aprire l'applicazione Blocco Note presumibilmente facendo clic sulla corrispondente icona con il mouse.
2. Abbiamo scritto il nostro codice sorgente all'interno dell'interfaccia grafica di Blocco Note utilizzando la tastiera.
3. Abbiamo salvato il file all'interno del nostro hard disk (memoria ausiliare), in una precisa cartella (`C:\ManualeDiJava9`).
4. Abbiamo aperto una sessione del programma "prompt dei comandi" invocando il comando "`CMD`" e ci siamo spostati nella cartella contenente il file appena salvato.
5. Quindi abbiamo compilato il nostro file invocando il programma **javac** contenuto nella cartella `bin` del JDK, scrivendo con la tastiera il comando "`javac HelloWorld.java`" direttamente sull'interfaccia grafica della prompt dei comandi.
6. Il programma compilatore è stato caricato nella memoria principale

(RAM) insieme al suo input (il file sorgente `HelloWorld.java`).

7. Il compilatore ha eseguito le sue istruzioni grazie al processore ed ha prodotto come output la traduzione in bytecode del codice sorgente, salvandolo in un file `HelloWorld.class`.
8. Abbiamo eseguito l'interprete definito nella Java Virtual Machine con il comando **java** contenuto nella cartella `bin` del JDK, scrivendo con la tastiera il comando "`java HelloWorld`" direttamente sull'interfaccia grafica della prompt dei comandi.
9. La JVM è stata caricata nella memoria principale (RAM) insieme al suo input (il file sorgente `HelloWorld.class`).
10. L'interprete Java ha tradotto le istruzioni Java in linguaggio macchina che è comprensibile al processore, e questo ha eseguito la stampa a video della scritta "`Hello World!`".

1.5 Possibili problemi in fase di compilazione ed esecuzione

Di solito, nei primi tempi, gli aspiranti programmatori Java ricevono spesso dei messaggi apparentemente misteriosi da parte del compilatore e dell'interprete Java. Non bisogna scoraggiarsi! Bisogna avere pazienza e imparare a leggere i messaggi che vengono restituiti. Inizialmente può sembrare difficile ma in breve tempo ci si accorge che gli errori che si commettono sono spesso gli stessi. Saper leggere gli errori del compilatore renderà voi programmatori migliori.

1.5.1 Possibili messaggi di errore in fase di compilazione

```
"javac" non è riconosciuto come comando interno o esterno,
un programma eseguibile o un file batch.
```

In questo caso non è il compilatore che ci sta segnalando un problema, bensì,

è lo stesso sistema operativo che non riconosce il comando `javac` che dovrebbe eseguire il compilatore del JDK. Sicuramente quest'ultimo non è stato installato correttamente. In questo caso è probabile che non sia stata impostata correttamente la variabile d'ambiente `PATH` (cfr. appendice B).

```
HelloWorld.java:1: error: class Helloworld is public, should
be
declared in a file named Helloworld.java
public class Helloworld {
       ^
1 error
```

In questo caso abbiamo chiamato il file `HelloWorld` e la classe `Helloworld` (notare le "w" minuscola). Il compilatore non è così intelligente da capire che non l'abbiamo fatto volontariamente.

```
HelloWorld.java:3: error: cannot find symbol
        System.out.printl();
                   ^
  symbol:   method printl()
  location: variable out of type PrintStream
1 error
```

Se riceviamo questo messaggio, abbiamo semplicemente scritto "`printl`" in luogo di "`println`". Il compilatore non può da solo accorgersi che è stato semplicemente un errore di battitura, ed allora ci ha segnalato che il metodo `printl()` non è stato trovato. È fondamentale comprendere i messaggi di errore che ci fornisce il compilatore, tenendo ben presente però, che ci sono dei limiti a questi messaggi ed alla comprensione degli errori del compilatore stesso.

```
HelloWorld.java:1: error: class, interface, or enum expected
public Class HelloWorld {
       ^
C:\Users\Claudio\Desktop\test\HelloWorld.java:2: error: class,
interface, or enum expected
    public static void main(String args[]) {
                   ^
C:\Users\Claudio\Desktop\test\HelloWorld.java:4: error: class,
interface, or enum expected
    }
    ^
```

```
3 errors
```

In questo caso non è vero che ci sono tre errori, in realtà l'errore è uno soltanto: abbiamo scritto `class` con lettera maiuscola e quindi la JVM ha richiesto esplicitamente una dichiarazione di classe (o di interfaccia o enum, concetti che chiariremo più avanti). "`Class`" lo ripetiamo, non è la stessa cosa di "`class`" in Java.

```
HelloWorld.java:3: error: ';' expected
        System.out.printl()
                           ^
1 error
```

In questo caso il compilatore ha capito subito che abbiamo dimenticato un punto e virgola che serve a concludere ogni statement (istruzione). Purtroppo il nostro compilatore non sarà sempre così preciso. In alcuni casi, se dimentichiamo un punto e virgola, o peggio dimentichiamo di chiudere un blocco di codice con una parentesi graffa, il compilatore ci potrebbe segnalare l'esistenza di errori inesistenti in righe di codice successive.

1.5.2 Possibili messaggi relativi alla fase di interpretazione

Solitamente in questa fase vengono sollevate dalla JVM quelle che vengono definite come "eccezioni". Alle eccezioni sarà dedicato un intero capitolo più avanti.

```
Errore: il metodo principale non è stato trovato nella classe
HelloWorld. Definire il metodo principale come:
   public static void main(String[] args)
```

Se questo è il messaggio di risposta ad un tentativo di mandare in esecuzione un programma, abbiamo probabilmente definito male il metodo `main()`. Probabilmente abbiamo dimenticato di dichiararlo `static` o `public`, oppure abbiamo scritto male la lista degli argomenti (che deve essere `String args[]`) o magari non abbiamo chiamato "`main`" il metodo.

```
Errore: impossibile trovare o caricare la classe principale
 HelloWorld.class
Causato da: java.lang.ClassNotFoundException: HelloWorld.class
```

In questo caso abbiamo provato ad eseguire il comando:
“`java HelloWorld.class`” invece di “`java HelloWorld`”.

> **Quando un programma viene eseguito, il tempo in cui è in esecuzione viene detto “runtime” (letteralmente “tempo di esecuzione”) del programma.**

1.6 Altri ambienti di sviluppo

Per ora abbiamo visto come si interagisce da riga di comando per invocare il compilatore con il comando `javac`, e l’interprete con il comando `java`. Per scrivere il nostro codice abbiamo utilizzato un semplice editor di testo come Blocco Note. Reputiamo importante capire come funziona Java a basso livello, e possiamo continuare a studiare l’intero libro utilizzando questi strumenti, ma è oggettivamente scomodo lavorare con due finestre diverse.

Quando inizieremo a programmare seriamente, probabilmente utilizzeremo un IDE come Netbeans o Eclipse. Questi fantastici strumenti però, non sono consigliati a chi sta iniziando a programmare. La loro complessità può distrarre dal vero obiettivo, ovvero quello di studiare Java. Se dobbiamo anche imparare ad utilizzare l’IDE potremmo non trovarci nelle condizioni ideali per concentrarci sul linguaggio. Inoltre utilizzare un IDE per i piccoli programmi che scriveremo inizialmente, è un po’ come sparare ad una mosca con un cannone. Infine le comodità offerte da un IDE come l’auto-completamento del codice o i suggerimenti possono causare dipendenza e pigrizia! Non è raro incontrare persone che programmano con questi strumenti da anni, senza avere chiari alcuni concetti basilari. Quindi se state iniziando a programmare sconsigliamo di utilizzare un IDE da subito. Nel capitolo 5 introdurremo altri ambienti di sviluppo tra cui gli IDE, spiegando come iniziare ad utilizzarli. Da quel punto in poi dovrebbe essere più facile iniziare a lavorare con gli IDE, perché dovreste già avere una certa confidenza con il linguaggio.

Ovviamente potreste non essere d'accordo e scegliere comunque di ignorare il consiglio di rimandare l'utilizzo di un IDE. In tal caso, cercate almeno di non dimenticare qual è il vostro primo obiettivo.

Chi vi scrive ha creato un editor Java open source pensato proprio per chi inizia a programmare: EJE (acronimo di Everyone's Java Editor). Non è un IDE, ma un semplice editor che è stato creato non per scrivere file di testo ma per programmare in Java. Offre alcune comodità rispetto ad un editor di testo generico. Tra le sue caratteristiche ricordiamo la colorazione della sintassi di Java, e il completamento di testo di alcune espressioni. Soprattutto è possibile compilare e mandare in esecuzione file, tramite la pressione di semplici bottoni, senza doversi spostare su una prompt dei comandi. L'utilizzo è davvero semplice e non c'è bisogno di studiarlo. È possibile eseguire il download gratuito dell'editor Java open source EJE, agli indirizzi http://www.claudiodesio.com/ e http://sourceforge.net/projects/eje/.

Riassumendo, se avete già una certa confidenza con ambienti di sviluppo integrati (IDE) allora potreste anche scegliere di utilizzare Netbeans o Eclipse. Se invece state iniziando a programmare solo ora, o comunque volete cercare di focalizzarvi solo sullo studio del linguaggio il consiglio è di utilizzare per i primi tempi EJE, per poi passare ad un IDE dopo il capitolo 5.

Riepilogo

L'**hardware** è la parte fisica del computer. Alcuni componenti hardware raccolgono **input** dall'esterno (mouse, tastiera...), altri forniscono gli **output** (monitor, stampante...), altri componenti non sono utilizzati direttamente dall'utente del computer (scheda madre, processore, memoria...). Il **processore** esegue operazioni matematiche e gestisce gli spostamenti dei dati in memoria. La **memoria** si divide in **memoria principale** (memoria RAM) dove vengono caricati i programmi (con i loro dati) che vengono mandati in esecuzione, e **memoria ausiliare** che a differenza della memoria principale sopravvive allo spegnimento del computer.

Il **software** è il termine che utilizziamo per riferirci ai programmi che eseguono su un computer. Il **sistema operativo** è un programma che gestisce nella sua interezza un computer, permettendo anche l'installazione e l'esecuzione di altri programmi.

Un **linguaggio di programmazione** ha sempre un vocabolario e un

insieme di regole. Un **processore** non può comprendere un linguaggio di programmazione perché è capace di utilizzare solo due simboli (`0` e `1`) e un particolare **linguaggio macchina** che utilizza istruzioni scritte con questi due simboli. Per programmare bisogna quindi scrivere il codice sorgente con il linguaggio di programmazione in file testuali, per poi tradurlo in linguaggio macchina tramite un software come un compilatore o un interprete. Un **compilatore** traduce le istruzioni scritte nel linguaggio di programmazione in istruzioni in linguaggio macchina creando **file binari** (eseguibili). Un **interprete** invece traduce le istruzioni contenute nel codice sorgente in istruzioni del codice macchina al volo.

Il termine "Java" si utilizza sia per riferirsi al linguaggio di programmazione, sia alla **tecnologia** da esso generata (che si dirama in tante sotto-tecnologie). Le sue caratteristiche principali sono la robustezza, la standardizzazione, la sintassi, la potenza e soprattutto l'**indipendenza della piattaforma.** Java è un linguaggio sia compilato che interpretato. La compilazione però non crea file contenenti istruzioni in linguaggio macchina, bensì file contenenti istruzioni scritte in un linguaggio vicino al linguaggio macchina chiamato **bytecode.** L'interprete Java definito all'interno della **Java Virtual Machine (JVM)** è capace di interpretare il bytecode con ottime performance.

Il **Java Development Kit (JDK)** è l'ambiente di sviluppo ufficiale di Java. Con un semplice editor di testo come Blocco Note, e la conoscenza basilare della prompt dei comandi, permette di programmare in Java. I passi da fare per poter scrivere ed eseguire un programma sono sempre fissi: scrivere il codice, salvarlo, compilarlo ed eseguire il programma.

Un programma Java si scrive utilizzando il concetto di classe, che contiene tutto il codice operativo del programma. Ogni classe deve essere scritta nel proprio **file sorgente** (con suffisso "`.java`"), e deve avere esattamente lo stesso nome del file. Un file che deve essere eseguito deve contenere un metodo chiamato **main**, che rappresenta il punto di partenza di un programma Java. Il processo di compilazione creerà un relativo file con suffisso "`.class`" che conterrà il bytecode. Questo file può essere dato in input ad una qualsiasi Java Virtual Machine di una qualsiasi piattaforma hardware-software per essere eseguito.

È particolarmente importante saper interpretare i **messaggi di errore** del

compilatore, e quelli lanciati dalla JVM durante l'esecuzione del programma.

È sconsigliato ai neofiti iniziare a programmare direttamente con un **IDE** per concentrarsi solo sullo studio del linguaggio. Conviene rimandare per quanto possibile l'utilizzo di IDE (almeno sino allo studio del capitolo 5) per non essere travolti anche dalla complessità dell'IDE stesso. Per i neofiti è stato creato appositamente EJE, un editor con funzionalità di base molto semplice da utilizzare.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Si tenga presente che gli esercizi sono parte integrante del testo e sono stati esternalizzati solo per minimizzare il numero di pagine e conseguentemente il costo di questo libro. Negli esercizi modificherete e creerete nuovo codice, confermerete le vostre conoscenze, e soprattutto imparerete cose nuove. Sembra strano ma capire la teoria perfettamente non significa automaticamente saper programmare. È incredibile quante difficoltà si incontrino a programmare anche quando le idee sembrano chiarissime. Senza la pratica non esiste programmazione Java.

Obiettivi del capitolo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper definire tutti i concetti informatici di base propedeutici allo studio della programmazione (unità 1.1)	☐	
Saper definire il linguaggio di programmazione Java e le sue caratteristiche (unità 1.2)	☐	
Interagire con l'ambiente di sviluppo: il Java Development Kit (unità 1.3, 1.4)	☐	

Saper digitare, compilare e mandare in esecuzione una semplice applicazione (unità 1.4, 1.5)	☐	
Aver scelto coscientemente un ambiente di sviluppo per lo studio iniziale di Java (unità 1.6)	☐	

Note:

2

Componenti fondamentali di un programma Java

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Comprendere cosa significa sviluppare un programma in Java, cosa sono i processi e i paradigmi di programmazione (unità 2.1).
- ✔ Saper definire i concetti di classe, oggetto, variabile, metodo e costruttore (unità 2.2, 2.3, 2.4, 2.5).
- ✔ Saper dichiarare una classe (unità 2.2).
- ✔ Istanziare oggetti da una classe (unità 2.2).
- ✔ Utilizzare i membri pubblici di un oggetto sfruttando l'operatore dot (unità 2.2, 2.3, 2.4).
- ✔ Dichiarare ed invocare un metodo (unità 2.3).
- ✔ Saper dichiarare e assegnare un valore ad una variabile (unità 2.4).
- ✔ Saper definire ed utilizzare i diversi tipi di variabili (d'istanza, locali e parametri formali) (unità 2.4).
- ✔ Dichiarare ed invocare un metodo costruttore (unità 2.5).
- ✔ Saper definire un costruttore di default (unità 2.5).
- ✔ Saper definire una classe appartenente ad un package e saper importare classi da altri package (unità 2.6).

- ✔ Avere un’idea di tutti gli altri componenti fondamentali della programmazione Java che studieremo nei prossimi capitoli (unità 2.7).

Da questo capitolo parte lo studio del linguaggio vero e proprio. Familiarizzeremo con i concetti fondamentali che sono alla base di un programma Java, sia in maniera teorica che pratica. Questo significa capire cosa scrivere all’interno dei nostri programmi.

Ecco una lista di concetti che sono alla base della conoscenza del linguaggio e che studieremo in questo capitolo:

- ❑ Classe
- ❑ Oggetto
- ❑ Variabile
- ❑ Metodo
- ❑ Costruttore
- ❑ Package

Esistono altri componenti importanti per programmare in Java, come per esempio le interfacce, le annotazioni, le classi innestate, le enumerazioni, i package, e la più grande novità della versione 9: i moduli. Per scelta didattica si è scelto di evitare di introdurre nozioni che attualmente non sono necessarie per iniziare a programmare. Questi altri concetti verranno presentati durante la lettura di questo libro nel momento in cui il lettore dovrebbe essere pronto per apprezzarne l’utilizzo.

Anche in questo capitolo non daremo nulla per scontato e quindi anche il lettore che non ha mai programmato dovrebbe essere messo nella condizione di poter iniziare. I lettori che invece hanno esperienza con la programmazione funzionale, troveranno profonde differenze con i concetti dell’object orientation. Il consiglio per questi ultimi è di non ancorarsi troppo a quello che già si conosce. È inutile per esempio forzare una classe ad avere il ruolo che poteva avere una funzione nella programmazione funzionale. Meglio far finta di partire da zero e non cercare scorciatoie che potrebbero portare ad un

apprendimento parziale degli argomenti.

2.1 Cosa significa creare un programma Java

Nel capitolo precedente abbiamo approcciato al codice Java creando un primo programma molto semplice. In realtà l'esempio del programma `HelloWorld` ci è servito più a prendere confidenza con l'ambiente di sviluppo che con il codice, perché quest'ultimo è stato spiegato solo parzialmente. Inoltre era un programma che non aveva le caratteristiche giuste: era costituito da un'unica classe con un metodo `main()` che stampava direttamente una frase a video. Scrivere un programma in Java invece significa creare diversi file (ognuno contenente solitamente un'unica classe). Potrebbero essere meno di dieci, ma anche decine, centinaia o anche migliaia se il progetto è vasto. Le classi non saranno isolate bensì collaboreranno per risolvere problemi, ognuna dovrebbe avere un ruolo preciso. Tra queste classi ne esisterà una sola che conterrà il metodo `main()`. Questa è una classe speciale dedicata a funzionare da punto di partenza dell'applicazione. Essa solitamente non eseguirà istruzioni direttamente (come la classe `HelloWorld` del capitolo precedente che stampava una stringa a video) ma istanzierà oggetti all'interno del metodo `main()`, i quali collaborando eseguiranno le istruzioni dei metodi di cui sono composti. Ogni classe dichiarerà al loro interno variabili, metodi e costruttori. I metodi conterranno le istruzioni da eseguire. Le classi infine saranno raggruppate in cartelle chiamate "package", per essere organizzate con un criterio logico.

Prima però di studiare i componenti fondamentali di un programma Java, dobbiamo capire alcuni concetti di base fondamentali sulla programmazione.

2.1.1 Programmazione e paradigmi

Lo **sviluppo del software**, è un processo creativo che porta a creare un programma partendo da un'idea. Questo **processo** è composto da una serie di attività da svolgere, ma non esiste un processo standard che sia perfetto per ogni situazione di programmazione. Quindi tali attività cambiano di tipologia, di ordine di esecuzione e di numero, a seconda del programma da creare.

Inoltre, di queste attività, solo alcune riguardano la programmazione. Il grande errore che commettono i programmatori poco esperti è quello di pensare che per programmare basti saper programmare. Saper programmare ci permetterà di risolvere gli esercizi che troverete nella prima parte di questo libro ma di certo non basterà per creare un vero programma partendo da zero. Per tramutare un'idea in un programma bisogna risolvere dei problemi concettuali, matematici, architetturali, funzionali e a volte filosofici. Potremmo anche essere così bravi a risolvere tutti questi problemi con la sola programmazione ma è molto probabile che alla fine la qualità del codice e la tempistica non risultino soddisfacenti.

I **processi** (a volte chiamati erroneamente **metodologie)**, di cui parleremo a partire dal quinto paragrafo, richiedono sempre delle fasi preliminari e propedeutiche alla programmazione. Se volessimo rappresentare un processo minimale, come esempio potremmo pensare che sia composto quantomeno dalle seguenti attività:

1. capire tutte le funzionalità che l'applicazione deve implementare (ovvero capire *cosa* deve fare il nostro programma);
2. trovare delle strategie per creare il programma nel modo migliore possibile (capire *come* implementare il nostro programma);
3. implementare con il codice le soluzioni ipotizzate (ovvero programmare);
4. testare il corretto funzionamento del nostro programma e risolvere eventuali problemi.

Questi ci sembrano i passi indispensabili che devono essere eseguiti per sviluppare qualsiasi tipo di software. Per ognuno di questi passi esistono tantissime tecniche che permettono di arrivare ad una soluzione; vedremo più avanti alcune di esse. Per ora dobbiamo tenere presente però che la programmazione è solo un pezzo del puzzle. Riuscite ad immaginare di poter scrivere un programma senza conoscere cosa deve fare (quindi saltando il passo 1 e di conseguenza il passo 2)? Beh, in realtà succede tantissime volte, e i risultati sono sempre disastrosi.

Quindi supponendo di trovarci nella situazione di aver eseguito correttamente il passo 1 e che ci siano chiare tutte le funzionalità da

implementare, come facciamo a capire qual è il modo migliore per implementarle?

Ogni linguaggio di programmazione offre determinati strumenti per programmare. In base a tali strumenti è possibile utilizzare un certo **paradigma di programmazione**, ovvero utilizzare dei processi che portano a creare programmi strutturati con determinate caratteristiche. In altre parole un paradigma di programmazione consente di creare un programma eseguendo delle attività ben definite, che faranno in modo che il nostro software possa godere dei vantaggi che offre il paradigma stesso. Ma quali sono i paradigmi esistenti? Ne esistono diversi, per ora ci concentreremo sul paradigma ad oggetti.

2.1.2 Le basi della programmazione ad oggetti (OOP)

Java è nato con l'intento di essere un **linguaggio orientato agli oggetti** (in inglese **object oriented)**, ovvero un linguaggio che usa il **paradigma ad oggetti.** Si dice anche che Java supporta la **programmazione ad oggetti** (in inglese **object oriented programming**, da cui l'acronimo **OOP).** In realtà negli ultimi anni l'evoluzione ha portato Java a supportare anche altri paradigmi di programmazione, ma per ora il nostro obiettivo è quello di capire come funziona la programmazione ad oggetti.

La programmazione ad oggetti è uno stile di programmazione che si basa su concetti ispirati dal mondo reale. Il nostro scopo quindi, è quello di capire bene questi concetti per poi riuscire ad utilizzarli *naturalmente* all'interno di un programma. Anche se i concetti basilari di classe ed oggetto saranno spiegati in dettaglio nel prossimo paragrafo, possiamo già introdurli prendendo spunto dal mondo reale. Per fare un esempio partiamo dal primo oggetto che vedete di fronte a voi: un libro. Il libro che avete davanti ha un titolo ("Manuale di Java 9"), un editore ("Hoepli"), un autore ("Claudio De Sio Cesari") e sicuramente sarete in grado da soli di elencare altre caratteristiche di questo libro come il costo, il numero di pagine e così via. Ad ognuna di queste caratteristiche sapreste anche assegnare un valore preciso. Ma "Manuale di Java 9" è solo uno dei tanti libri che esistono. Se invece chiudete gli occhi e pensate a cos'è un libro, non pensate a "Manuale di Java 9" e a nessun altro libro in particolare, ma semplicemente ad un oggetto che abbia delle pagine da leggere. La differenza tra il "Manuale di

Java 9” e la vostra idea di libro, è che quest’ultima non esiste nella realtà: l’idea non si può leggere! È solo un’astrazione, un’idealizzazione di tanti oggetti che esistono nella realtà: i libri. Un’idea non esiste nel mondo reale, non possiamo toccarla con mano. Nella nostra mente un libro ha sicuramente un numero di pagine ma non ha un numero di pagine specifico. Invece “Manuale di Java 9” è un oggetto libro concreto con un numero preciso di pagine.

Ecco, ora possiamo già capire i due concetti che sono alla base dell’OOP: la classe e l’oggetto. L’idea di libro che abbiamo nella nostra mente è rappresentabile in Java con una classe `Libro`. Una classe è appunto solo un concetto, un’astrazione, un’idea, e ci serve solamente per definire come saranno fatti gli oggetti veri e propri. Quindi potremo creare all’interno dei nostri programmi degli oggetti libro concreti partendo dal modello della classe `Libro`.

Così come un’idea di libro rappresenta un’astrazione di una serie di oggetti concreti che sono i libri che esistono nella realtà, in Java una classe rappresenta un’astrazione (che fa da modello) per gli oggetti libri che possono essere creati ed utilizzati dal programma. Quindi in Java possiamo definire la classe `Libro` ed usarla come modello per creare oggetti specifici come l’oggetto `manualeDiJava9`. Cerchiamo ora di entrare più nel dettaglio.

2.2 Le basi della programmazione object oriented: classi ed oggetti

Un linguaggio orientato agli oggetti è un linguaggio che supporta i concetti di classe ed oggetto. Questi due concetti sono strettamente legati tra loro. Se volessimo definire formalmente questi due concetti, dovremmo asserire:

- definizione 1:
  - una **classe** è un’astrazione indicante un insieme di oggetti che condividono le stesse caratteristiche e le stesse funzionalità;

- definizione 2:
  - un **oggetto** è un'istanza (ovvero una creazione fisica) di una classe.

Cerchiamo di capire meglio queste due definizioni. Ogni concetto che fa parte della teoria dell'Object Orientation infatti, esiste anche nel mondo reale. Questa teoria è nata proprio per soddisfare l'esigenza umana di interagire con il software in maniera più naturale (cfr. paragrafo 6.1). Si tratterà quindi solo di associare la giusta idea al nuovo termine. Passiamo subito ad un esempio pratico:

```
public class Punto
{
    public int x;
    public int y;
}
```

Con Java è possibile creare codice per astrarre un qualsiasi concetto del mondo reale. Abbiamo appena definito una classe `Punto`. Evidentemente lo scopo di questa classe è definire il concetto di punto (a due dimensioni) tramite la definizione delle sue coordinate su di un piano cartesiano. Questo codice è abbastanza intuitivo da essere compreso anche senza tante spiegazioni. Le coordinate `x` e `y` sono state astratte mediante due **attributi** (dette solitamente **variabili)** dichiarati di tipo intero `(int)`. Questo significa che potranno assumere solo valori di tipo numero intero. Per esempio sarà possibile assegnare alla variabile `x` il valore `5`, ma non il valore `abc` o il valore `5.2`, visto che questi ultimi non sono numeri interi. Le variabili `x` e `y` sono anche state dichiarate pubbliche (`public`), il che significa che potrebbero essere utilizzate anche da altre classi diverse dalla classe `Punto`. Possiamo immaginare che questa classe costituisca una parte di un'applicazione Java, magari un'applicazione di disegno. Nel paragrafo precedente abbiamo visto che possiamo salvare questo listato in un file chiamato `Punto.java` e compilarlo tramite il comando:

```
javac Punto.java
```

ottenendo così il file `Punto.class`. Questa volta però non possiamo mandarlo in esecuzione tramite il comando:

```
java Punto
```

Otterremmo infatti il seguente messaggio di errore:

```
Errore: il metodo principale non è stato trovato nella classe
Punto.
Definire il metodo principale come:
    public static void main(String[] args)
```

Infatti in questa classe è assente il metodo `main()` (cfr. capitolo precedente), che abbiamo definito come punto di partenza dell'esecuzione di ogni applicazione Java e che avrebbe reso la classe una classe speciale. Se eseguiamo il comando `java Punto`, la JVM prova a cercare questo metodo per eseguirlo e non trovandolo termina istantaneamente l'esecuzione. Ma davvero desidereremmo una classe che si chiama `Punto` che esegue del codice? Nel mondo reale *l'idea di un punto su un piano cartesiano* esegue qualcosa? Non sembra una buona idea creare una classe chiamata `Punto` che rappresenti il punto di partenza di un'applicazione, perché nel mondo reale l'idea di punto non sembra affatto un'entità che esegue delle attività. Ciò è assolutamente in linea con la definizione di classe (def. 1). Infatti, se volessimo trovare nel mondo reale un sinonimo di classe, dovremmo pensare a termini come *idea, astrazione, concetto, modello* o *definizione*. Quindi, definendo una classe `Punto` con codice Java abbiamo solo definito all'interno del nostro futuro programma il concetto di punto, secondo la nostra interpretazione e nel contesto in cui ci vogliamo calare (per esempio un programma di disegno). Con il codice scritto finora non abbiamo ancora definito nessun punto vero e proprio (per esempio il punto di coordinate `x=5` e `y=6`).

Quindi, come nel mondo reale, se non esistono punti concreti (ma solo la definizione di punto) non può accadere nulla che abbia a che fare con un punto. Ovvio anche che nel codice Java la classe `Punto` non dovrebbe essere eseguibile da sola!

Occorre quindi definire gli *oggetti punto*, ovvero le creazioni fisiche

realizzate a partire dalla definizione data dalla classe.

Nel contesto della programmazione ad oggetti una classe dovrebbe limitarsi a definire che struttura avranno gli oggetti che da essa saranno istanziati. **Istanziare**, come abbiamo già affermato, è il termine object oriented che si usa in luogo di *creare fisicamente*, ed il risultato di un'istanziazione viene chiamato **oggetto.**

Per esempio, istanzieremo oggetti dalla classe `Punto` creando un'altra classe che contiene un metodo `main()` che chiameremo `TestOggettiPunto`:

```
1 public class TestOggettiPunto
2 {
3     public static void main(String args[])
4     {
5         Punto punto1;
6         punto1 = new Punto();
7         punto1.x = 2;
8         punto1.y = 6;
9         Punto punto2 = new Punto();
10        punto2.x = 0;
11        punto2.y = 1;
12        System.out.println(punto1.x);
13        System.out.println(punto1.y);
14        System.out.println(punto2.x);
15        System.out.println(punto2.y);
16    }
17 }
```

Analizziamo la classe `TestOggettiPunto`. Diamo per scontato che sia chiaro che alla riga 1 è stata dichiarata la classe pubblica `TestOggettiPunto`, che termina alla riga 17, e che alla riga 3 è stato dichiarato il metodo `main()` che termina alla riga 16. Dovrebbe essere altrettanto chiaro ormai che le parentesi graffe delimitano le strutture di classe e di metodo, e per tale ragione sono state allineate (indentate) opportunamente, per rendere esplicito il loro abbinamento.

Una riga di codice viene solitamente chiamata **statement** e può contenere una o più istruzioni. All'interno del metodo `main()` troviamo gli statement che devono essere eseguiti, dalla riga 5 alla riga 15. Ogni statement in Java termina sempre con un ";".

Alla riga 5 troviamo la prima istruzione da eseguire: la **dichiarazione di**

un oggetto di tipo `Punto` che viene chiamato `punto1`:

```
5       Punto punto1;
```

Dichiarare un oggetto non significa crearlo ma solo definire il suo nome ed associargli un tipo. La sintassi per dichiarare un oggetto consiste nel far precedere al nome dell'oggetto (in questo caso `punto1`) la dichiarazione del suo tipo (la classe di tipo `Punto`).

Alla riga 6 invece troviamo la sintassi che crea l'oggetto `punto1` della classe `Punto`:

```
6       punto1 = new Punto();
```

In questo statement ci sono due istruzioni. Infatti con l'istruzione `new Punto()` viene istanziato l'oggetto mentre l'operatore `=` assegna l'oggetto creato al "nome" `punto1`. Quindi la parola chiave `new` istanzia la classe `Punto` a cui viene assegnato il nome `punto1`.

Dalla riga 6 in poi è quindi possibile utilizzare l'oggetto `punto1`. Precisamente alle righe 7 e 8, assegniamo alle coordinate `x` e `y` del `punto1` rispettivamente i valori interi `2` e `6`:

```
7       punto1.x = 2;
8       punto1.y = 6;
```

Sfruttando la definizione che ci ha fornito la classe `Punto`, abbiamo creato un oggetto di tipo `Punto` che è individuato dal nome `punto1`. Notiamo l'utilizzo dell'**operatore dot** (che in inglese significa *punto*, ma nel senso del simbolo di punteggiatura ".") per accedere alle variabili `x` e `y`. L'istruzione `punto1.x = 2`, per esempio, assegna il valore intero `2` alla variabile `x` dell'oggetto `punto1`. Notare come le variabili `x` e `y` siano state dichiarate nella classe `Punto`, ma siano valorizzate per l'oggetto `punto1`, e quindi appartengono all'oggetto `punto1`. La classe `Punto` ci è servita solo come modello per istanziare `punto1`. Notiamo inoltre che all'interno della classe `Punto` le variabili `x` e `y` erano state dichiarate di tipo intero (`int`).

Ricapitolando, abbiamo prima dichiarato l'oggetto alla riga 5, l'abbiamo istanziato alla riga 6, e l'abbiamo utilizzato (impostando le sue variabili) alle righe 7 e 8.

Alla riga 9 poi, abbiamo dichiarato ed istanziato con un'unica riga di codice un altro oggetto dalla classe `Punto`, chiamandolo `punto2`:

```
 9        Punto punto2 = new Punto();
```

Tecnicamente le istruzioni alla riga 9 erano tre: dichiarazione dell'oggetto `punto2`, istanza di un oggetto `Punto`, e assegnazione dell'oggetto appena creato a `punto2`. Abbiamo poi impostato le coordinate di quest'ultimo a `0` e `1`:

```
 10       punto2.x = 0;
 11       punto2.y = 1;
```

Abbiamo infine stampato le coordinate di entrambi i punti:

```
 12       System.out.println(punto1.x);
 13       System.out.println(punto1.y);
 14       System.out.println(punto2.x);
 15       System.out.println(punto2.y);
```

Le definizioni di classi ed oggetto dovrebbero essere un po' più chiare: la classe è servita per definire come saranno fatti gli oggetti. L'oggetto rappresenta una realizzazione fisica della classe. In questo esempio abbiamo istanziato due oggetti diversi da una stessa classe. Entrambi questi oggetti sono punti, ma evidentemente sono punti diversi (si trovano in diverse posizioni sul riferimento cartesiano).

> **Questo ragionamento, in fondo, è già familiare a chiunque giacché derivante dal mondo reale che ci circonda. L'essere umano, per superare la complessità della realtà, raggruppa gli oggetti in classi. Per esempio, nella nostra mente esiste un modello definito dalla classe `Persona` anche se nella realtà esistono miliardi di oggetti di tipo `Persona` ognuno dei quali ha caratteristiche uniche.**

2.2.1 Osservazione importante sulla classe `Punto`

Nell'esempio precedente abbiamo potuto commentare la definizione di due

classi. Per la prima (la classe `Punto`) abbiamo sottolineato la caratteristica di rappresentare **un dato.** Tale classe è da considerarsi una parte strutturale dell'applicazione e quindi svolge un ruolo essenziale nell'ambito della compilazione. Nell'ambito dell'esecuzione dell'applicazione la classe `Punto` non ha un ruolo attivo. Infatti sono gli oggetti istanziati da essa che influenzano il flusso di lavoro del programma. Questo può definirsi come il caso standard. In un'applicazione object oriented una classe dovrebbe limitarsi a definire la struttura comune di un insieme di oggetti, e non dovrebbe mai "possedere" né variabili né metodi, ma solo dichiararli. Infatti, la classe `Punto` non possiede le variabili `x` e `y`, bensì, dichiarando le due variabili, definisce gli oggetti che da essa saranno istanziati come possessori di quelle variabili. Si noti che mai all'interno del codice dell'ambito di esecuzione del programma è presente un'istruzione come:

```
Punto.x
```

ovvero

```
NomeClasse.nomeVariabile
```

(che tra l'altro produrrebbe un errore in compilazione) bensì:

```
punto1.x
```

ovvero

```
nomeOggetto.nomeVariabile
```

L'operatore "." è sinonimo di "appartenenza". Sono quindi gli oggetti a possedere le variabili dichiarate nella classe (che chiameremo "variabili d'istanza", ovvero "variabili dell'oggetto"). Infatti, i due oggetti istanziati avevano valori diversi per `x` **e** `y`**, il che significa che ognuno dei due oggetti** `punto1` **e** `punto2` **ha la sua variabile** `x` **e la sua variabile** `y`**. Le variabili di** `punto1` **sono assolutamente indipendenti dalle variabili di** `punto2`**. Giacché le classi non hanno membri (variabili e metodi), non eseguono codice e non hanno un ruolo nell'ambito dell'esecuzione di un programma, e per**

quanto visto sono gli oggetti i protagonisti assoluti di quest'ambito. In pratica il rapporto tra classe ed oggetto è esattamente lo stesso che c'è tra idea e oggetto concreto nel mondo reale.

2.2.2 Osservazione importante sulla classe `TestOggettiPunto`

Come spesso accade quando si approccia Java, appena definita una nuova regola, subito si presenta l'eccezione (che se vogliamo... dovrebbe confermarla!). Puntiamo infatti la nostra attenzione sulla classe `TestOggettiPunto`. È una classe che esegue codice contenuto all'interno dell'unico metodo `main()`, che per quanto detto è assunto per default quale punto di partenza di un'applicazione Java. In qualche modo, infatti, i creatori di Java dovevano stabilire un modo per far partire l'esecuzione di un programma. La scelta è stata compiuta in base ad una questione pratica e storica: un'applicazione scritta in C o C++ ha come punto di partenza per default un cosiddetto "programma chiamante" che deve chiamarsi proprio `main()`. In Java i programmi chiamanti non esistono, ma esistono i metodi, ed in particolare i metodi statici, ovvero dichiarati con un modificatore `static`. Questo modificatore sarà trattato in dettaglio nel capitolo 6. Per ora limitiamoci a sapere che un membro dichiarato statico appartiene alla classe e che tutti gli oggetti istanziati da essa condivideranno gli stessi membri statici. Concludendo, giacché la classe `TestOggettiPunto` contiene il metodo `main()`, può eseguire del codice.

In ogni applicazione Java deve esserci una classe che contiene il metodo `main()`. Questa classe dovrebbe avere il nome dell'applicazione stessa, astraendo il flusso di lavoro che deve essere eseguito. In linea teorica, quindi, la classe contenente il metodo `main()` non dovrebbe contenere altri membri. Solitamente in un vero programma, il metodo `main()` crea oggetti che si occupano di eseguire istruzioni invece di eseguire istruzioni esso stesso.

2.2.3 Un’altra osservazione importante

Abbiamo presentato l’esempio precedente per la sua semplicità. È infatti *matematico* pensare ad un punto sul piano cartesiano come formato da due coordinate chiamate `x` e `y`. Ci sono concetti del mondo reale la cui astrazione non è così standard. Per esempio, volendo definire l’idea (astrazione) di un’auto, dovremmo parlare delle caratteristiche e delle funzionalità comuni ad ogni auto. Ma, se confrontiamo l’idea (ovvero la classe) di auto che definirebbe un intenditore di automobili con quella di una persona che non ha neanche la patente, ci sarebbero significative differenze. La persona non patentata potrebbe definire un’auto come “un mezzo di trasporto con quattro ruote ed un motore”, ma l’intenditore di automobili potrebbe dare una definizione alternativa molto più dettagliata definendo caratteristiche come assicurazione, telaio, modello, pneumatici etc. Entrambe queste definizioni potrebbero essere portate nella programmazione sotto forma di classi. Per convenienza ragioneremo *come se non avessimo la patente*, creando una classe che dichiara come attributo un intero chiamato `numeroRuote` inizializzato a `4` (supponendo che le auto abbiano tutte quattro ruote). Inoltre per semplicità definiamo un’altra variabile intera `cilindrata` (in luogo del motore) ed un metodo che potremmo chiamare `muoviti()`.

> **Nei prossimi paragrafi i metodi saranno definiti in dettaglio. Per adesso ci basta sapere che il metodo serve a definire una funzionalità che deve avere il concetto che si sta astraendo con la classe.**

```
public class Auto
{
    public int numeroRuote = 4;
    public int cilindrata; // quanto vale?
    public void muoviti()
    {
       // implementazione del metodo...
    }
}
```

Il codice che scriviamo dopo due slash (come nel metodo `muoviti()`), è un commento e non verrà preso in considerazione dal compilatore. I commenti di Java verranno comunque spiegati in dettagli nel prossimo capitolo.

Ogni auto ha 4 ruote (di solito), si muove ed ha una cilindrata (il cui valore non è definibile a priori). Una Ferrari California e una Fiat 600 hanno entrambe 4 ruote, una cilindrata e si muovono, anche se in modo diverso. La Ferrari e la Fiat sono da considerarsi oggetti della classe `Auto`, e nella realtà esisterebbero come oggetti concreti. Nella seguente classe vengono creati due oggetti della classe `Auto`, viene impostata loro la cilindrata opportunamente, e per entrambi gli oggetti verrà invocato il metodo `muoviti()`.

```
1       public class TestOggettiAuto
2       {
3           public static void main(String args[])
4           {
5               Auto fiat600;
6               fiat600 = new Auto();
7               fiat600.cilindrata = 1100;
8               fiat600.muoviti();
9               Auto california = new Auto();
10              california.cilindrata = 4300;
11              california.muoviti();
12          }
13      }
```

Dovrebbe essere abbastanza chiaro che i due oggetti si muoveranno entrambi ma con velocità diverse.

2.3 I metodi in Java

Nella definizione di classe, quando si parla di *caratteristiche* ci si riferisce ai **dati** (variabili e costanti) mentre col termine *funzionalità* ci si riferisce ai **metodi.** Infatti avevamo già accennato al fatto che il termine *metodo* è sinonimo di *azione*. Quindi, affinché un programma esegua qualche

istruzione deve contenere metodi. Per esempio è il metodo `main()` che, per default, è il punto di partenza di ogni applicazione Java. Una classe senza metodo `main()`, come la classe `Punto`, non può essere mandata in esecuzione ma solo istanziata all'interno di un metodo di un'altra classe (nell'esempio precedente nel metodo `main()` della classe `TestOggettiPunto`). Il concetto di metodo è quindi anch'esso alla base della programmazione ad oggetti. Senza metodi, gli oggetti non potrebbero comunicare tra loro e i programmi non potrebbero eseguire azioni. Inoltre i metodi rendono i programmi più leggibili e di più facile manutenzione, lo sviluppo più veloce e stabile, evitano le duplicazioni e favoriscono il riuso del codice.

Il programmatore che si avvicina a Java dopo esperienze nel campo della programmazione funzionale, spesso tende a confrontare il concetto di funzione con il concetto di metodo. Sebbene simili nella forma e nella sostanza, bisognerà tener presente che un metodo ha un ruolo differente rispetto ad una funzione. Nella programmazione funzionale infatti, il concetto di funzione era la base, come nella programmazione ad oggetti il concetto di classe ed oggetto sono la base. Tutti i programmi funzionali hanno un … … programma chiamante e un certo numero di funzioni. Queste avevano di fatto il compito di risolvere determinati sotto-problemi generati da un'analisi di tipo top-down, allo scopo di risolvere il problema generale. Come vedremo più avanti, nella programmazione orientata agli oggetti, i sotto-problemi saranno invece risolti tramite l'astrazione di classi ed oggetti, che a loro volta definiranno metodi. Quindi nonostante i metodi e le funzioni siano sostanzialmente molto simili, si differenziano per il loro ruolo all'interno del relativo paradigma di programmazione.

È importante cominciare a distinguere nettamente due fasi per quanto riguarda i metodi: dichiarazione e chiamata (ovvero la definizione e l'utilizzo).

2.3.1 Dichiarazione di un metodo

La dichiarazione definisce un metodo. Ecco la sintassi:

```
[modificatori] tipo_di_ritorno nome_del_metodo ([parametri])
{corpo_
del_metodo}
```

dove:

- **modificatori:** parole chiave di Java che possono essere usate per modificare in qualche modo le funzionalità e le caratteristiche di un metodo. Tutti i modificatori sono trattati in maniera approfondita nei prossimi capitoli. Esempi di modificatori sono `public` e `static`;
- **tipo di ritorno:** il tipo di dato che un metodo potrà restituire dopo essere stato chiamato. Questo potrebbe coincidere sia con un tipo di dato primitivo come un `int`, sia con un tipo complesso (un oggetto) come una stringa (definita dalla classe `String`). È anche possibile specificare che un metodo non restituisca nulla (`void`), come nel caso del `main()`;
- **nome del metodo:** identificatore del metodo;
- **parametri:** dichiarazioni di variabili che potranno essere passate al metodo come input e di conseguenza essere sfruttate nel corpo del metodo al momento della chiamata. Il numero di parametri può essere zero ma anche maggiore di uno. Se si dichiarano più parametri le loro dichiarazioni saranno separate da virgole;
- **corpo del metodo:** insieme di istruzioni (statement) che verranno eseguite quando il metodo sarà invocato.

> **La coppia costituita dal nome del metodo e l'eventuale lista dei parametri viene detta "firma" (in inglese "signature") del metodo.**

Per esempio, viene di seguito presentata una classe chiamata `Aritmetica` che

dichiara un banale metodo di somma di due numeri:

```
public class Aritmetica
{
    public int somma(int a, int b)
    {
        return (a + b);
    }
}
```

Si noti che il metodo presenta come modificatore la parola chiave `public`. Si tratta di uno specificatore (o modificatore) d'accesso che abbiamo già incontrato nella dichiarazione di classi. Se il metodo `somma()` è dichiarato `public` diventerà accessibile da altre classi. Precisamente, i metodi di altre classi potranno, dopo aver istanziato un oggetto della classe `Aritmetica`, invocare il metodo `somma()`. Esistono anche altri specificatori d'accesso come `private` sui quali approfondiremo il discorso nel capitolo 5.

Il tipo di ritorno è un `int`, ovvero un intero. Ciò significa che quando questo metodo sarà invocato restituirà in output un tipo di valore intero, ovvero che avrà come ultima istruzione un comando (`return`) che restituirà un numero intero. Inoltre questo metodo dichiara come lista di parametri due interi (chiamati `a` e `b`). I parametri di un metodo rappresentano l'input del metodo. Quando questo metodo verrà chiamato, bisognerà passargli in input due valori di tipo intero.

All'interno del blocco di codice delimitato dalle parentesi graffe, i due parametri saranno sommati con l'operatore `+` e la loro somma verrà restituita come output tramite il comando `return`. Nell'unico statement contenuto nel blocco di codice quindi ci sono due istruzioni: la somma dei due parametri e il comando `return`. Si noti che per l'istruzione di somma è stata circondata da due parentesi tonde. Anche se in questo caso non erano necessarie per la corretta esecuzione del metodo, abbiamo preferito evidenziare che la somma sarebbe stata eseguita prima dell'istruzione `return`. La somma tra due numeri interi non può essere che un numero intero, quindi è corretto dichiarare come tipo di ritorno il tipo `int`. Concludendo, la dichiarazione di un metodo definisce una lista di modificatori, un tipo di ritorno, un nome (identificatore), una lista di parametri e un blocco di codice che contiene le istruzioni che saranno eseguite quando il metodo sarà chiamato. La

definizione di un metodo però, fa parte della definizione di una classe, che come abbiamo visto non esegue codice. Per poter eseguire le istruzioni contenute in un metodo, bisogna creare un oggetto della classe che lo dichiara e invocarlo come mostrato nel prossimo paragrafo.

2.3.2 Chiamata (o invocazione) di un metodo

Di seguito presentiamo un'altra classe eseguibile (ovvero contenente il metodo `main()`) che istanzia un oggetto dalla classe `Aritmetica` e chiama il metodo `somma()`:

```
1 public class TestSomma
2 {
3     public static void main(String args[])
4     {
5         Aritmetica oggettoAritmetica = new Aritmetica();
6         int risultato = oggettoAritmetica.somma(5, 6);
7         System.out.println(risultato);
8     }
9 }
```

Concentriamoci direttamente sugli statement del metodo `main()`. Alla riga 5 viene istanziato un oggetto chiamato `oggettoAritmetica` di tipo `Aritmetica`. Tramite questo oggetto, alla riga 6, viene invocato il metodo `somma()` che abbiamo dichiarato nella classe `Aritmetica`. In questo caso notiamo subito che l'accesso al metodo `somma()` è avvenuto sempre tramite l'operatore *dot* come nel caso dell'accesso alle variabili. Quindi, tutti i membri (attributi e metodi) pubblici definiti all'interno di una classe, saranno accessibili tramite un'istanza della classe stessa che utilizza l'operatore dot. La sintassi è la seguente:

```
nomeOggetto.nomeMetodo();
```

e

```
nomeOggetto.nomeAttributo;
```

L'accesso al metodo di un oggetto provoca l'esecuzione del relativo blocco di codice. In quest'esempio poi, abbiamo definito una variabile intera `risultato` che ha immagazzinato il risultato della somma. Se non avessimo

fatto ciò, non avrebbe avuto senso chiamare un metodo che restituisce un valore dal momento che non l'avremmo potuto utilizzare in qualche modo! Invece dopo averlo immagazzinato nella variabile `risultato` stampiamo a video il suo valore, per verificare in fase di runtime del programma la reale esecuzione della somma. Per farlo abbiamo dovuto passare la variabile `risultato` come parametro al metodo che abbiamo già incontrato nel capitolo precedente: `System.out.println()`.

> **L'istruzione** `System.out.println(risultato)` **è anch'essa una chiamata ad un metodo. È un metodo che non abbiamo scritto noi ma che esiste nella sterminata libreria di classi che ci mette a disposizione Java. Tenendo presente quello che abbiamo detto sui metodi, sugli oggetti e sull'operatore dot, è possibile intuire che il metodo** `println()` **può prendere un parametro in input e che non restituisce un risultato in output. Ma sappiamo che i metodi vengono chiamati sugli oggetti, quindi** `System.out` **deve essere un oggetto. Infatti lo è, e rappresenta l'output standard del sistema, che per il sistema operativo Windows coincide con la prompt dei comandi. Ecco perché la chiamata su quest'oggetto del metodo** `println()` **provoca la scrittura sulla prompt dei comandi del parametro che gli viene passato in input. Più avanti studieremo più in dettaglio l'oggetto** `System.out`.

Da notare che avremmo anche potuto sostituire le righe 6 e 7 con la seguente:

```
  System.out.println(oggettoAritmetica.somma(5,6));
```

Con questo statement avremmo stampato al volo il valore di ritorno della chiamata al metodo `somma()` sull'oggetto `oggettoAritmetica`, che poi è lo stesso valore della variabile `risultato`.

Esistono anche metodi che non hanno parametri in input. Per esempio un metodo (inutile) che somma sempre gli stessi numeri e quindi restituisce sempre lo stesso valore come il seguente:

```
public class AritmeticaFissa
{
    public int somma()
    {
        return (5 + 6);
    }
}
```

oppure metodi che non hanno tipo di ritorno e quindi dichiarano come tipo di ritorno `void` (vuoto) come il metodo `main()`. Infatti il metodo `main()` è il punto di partenza del runtime di un'applicazione Java, quindi non deve restituire un risultato, non venendo chiamato esplicitamente da un altro metodo.

Esistono ovviamente anche metodi che non hanno né parametri in input, né in output (tipo di ritorno `void`), come per esempio un semplice metodo che visualizza a video sempre lo stesso messaggio:

```
public class Saluti
{
    public void stampaSaluto()
    {
        System.out.println("Ciao");
    }
}
```

Segue una banale classe che istanzia un oggetto dalla classe `Saluti` e chiama il metodo `stampaSaluto()`. Come si può notare, se il metodo ha come tipo di ritorno `void`, non bisogna "catturarne" il risultato.

```
public class TestSaluti
{
    public static void main(String args[])
    {
        Saluti oggetto1 = new Saluti();
        oggetto1.stampaSaluto();
    }
}
```

2.3.3 Varargs

È possibile anche utilizzare metodi che dichiarano come lista di parametri i cosiddetti **variable arguments**, più brevemente noti come **varargs.** In pratica

è possibile creare metodi che dichiarano un numero non definito di parametri di un certo tipo. La sintassi è un po' particolare, e fa uso di tre puntini (come i puntini sospensivi), dopo la dichiarazione del tipo. Per esempio, è possibile invocare il seguente metodo:

```
public class AritmeticaVariabile
{
    public void somma(int... interi)
    {
        //codice complicato...
    }
}
```

in uno qualsiasi dei seguenti modi:

```
AritmeticaVariabile ogg = new AritmeticaVariabile();
ogg.somma();
ogg.somma(1,2);
ogg.somma(1,4,40,27,48,27,36,23,45,67,9,54,66,43);
ogg.somma(1);
```

Quindi possiamo passare da 0 a n argomenti. Notare che per ogni dichiarazione di metodo è possibile dichiarare un solo parametro varargs come ultimo argomento. Quindi le seguenti dichiarazioni non compilerebbero:

```
public void somma(int... interi, int... altriInteri)
{
    //...
}
...
public void somma(int... interi, int altroIntero)
{
    //...
}
```

2.4 Le variabili in Java

Nella programmazione una **variabile** è una porzione di memoria in cui è immagazzinato un certo tipo di dato. Abbiamo sino ad ora incontrato come tipo di dato semplice l'intero `int`, con cui abbiamo dichiarato le variabili `x` e

`y` della classe `Punto`. I tipi di Java saranno approfonditi nel prossimo capitolo.

Anche per l'utilizzo delle variabili possiamo distinguere due fasi: **dichiarazione** ed **assegnazione.** L'assegnazione di un valore ad una variabile è un'operazione che si può ripetere molte volte nell'ambito dell'esecuzione del programma. La prima assegnazione viene chiamata spesso inizializzazione della variabile.

2.4.1 Dichiarazione di una variabile

La sintassi per la dichiarazione di una variabile è la seguente:

```
[modificatori] tipo_di_dato nome_della_variabile [ =
inizializzazione];
```

dove:

- **modificatori:** parole chiavi di Java che possono essere usate per modificare in qualche modo le funzionalità e le caratteristiche della variabile;
- **tipo di dato:** il tipo di dato della variabile;
- **nome della variabile:** identificatore della variabile;
- **inizializzazione:** valore di default con cui è possibile valorizzare una variabile.

Segue una classe che definisce in maniera superficiale un rettangolo:

```
public class Rettangolo
{
    public final int NUMERO_LATI = 4;
    public int altezza;
    public int larghezza;
}
```

In questo caso abbiamo definito due variabili intere chiamate `altezza` e `larghezza`, ed una costante `NUMERO_LATI`. Il modificatore `final` infatti (che sarà trattato in dettaglio nel capitolo 6) rende una variabile costante nel suo valore. Questo significa che se nel nostro programma un'istruzione tentasse

di cambiare il valore della costante, otterremo un errore in compilazione. È anche possibile definire due variabili dello stesso tipo con un'unica istruzione come nel seguente esempio:

```
public class Rettangolo
{
    public final int NUMERO_LATI = 4;
    public int altezza, larghezza;
}
```

In questo caso possiamo notare una maggiore compattezza del codice ottenuta evitando le duplicazioni, ma una minore leggibilità.

La dichiarazione di una variabile è molto importante. La posizione della dichiarazione definisce lo **scope**, ovvero la visibilità, ed il ciclo di vita della variabile stessa. In pratica potremmo definire tre diverse tipologie di variabili, basandoci sul posizionamento della dichiarazione: **variabili d'istanza, variabili locali** e **parametri formali.**

2.4.2 Variabili d'istanza

Una variabile è detta **variabile d'istanza** (o anche **variabile membro** o **attributo)** se è dichiarata in una classe ma al di fuori di un metodo.

> **Non è possibile comunque definire una variabile al di fuori di una classe.**

Le variabili definite nella classe `Rettangolo` sono tutte di istanza. Esse condividono il proprio ciclo di vita con l'oggetto (istanza) cui appartengono. Quando un oggetto della classe `Rettangolo` è istanziato, viene allocato spazio in memoria per tutte le sue variabili d'istanza che vengono inizializzate ai relativi valori nulli. Nel nostro caso, le variabili `altezza` e `larghezza` saranno inizializzate al valore `0` in quanto numeri interi. `NUMERO_LATI` invece è una costante esplicitamente inizializzata a `4` ed il suo valore non potrà cambiare. Una variabile d'istanza smetterà di esistere quando smetterà di esistere l'oggetto a cui appartiene.

2.4.3 Variabili locali

Una variabile è detta **locale** (o **di stack**, o **automatica**, o anche **temporanea)** se è dichiarata all'interno di un metodo. Essa smetterà di esistere quando terminerà il metodo.

Nel seguente esempio (ottenuto modificando il metodo `somma()` della classe `Aritmetica` incontrato precedentemente) la variabile `z` è una variabile locale:

```
public int somma(int x, int y)
{
    int z = x + y;
    return z;
}
```

Una variabile di questo tipo, a differenza di una variabile di istanza, non sarà inizializzata ad un valore nullo al momento dell'istanza dell'oggetto a cui appartiene. Le variabili locali non vengono inizializzate automaticamente quindi bisogna farlo esplicitamente. Infatti il compilatore potrebbe restituire un messaggio di errore laddove ci sia possibilità che la variabile non sia inizializzata al runtime.

2.4.4 Parametri formali

Le variabili dichiarate all'interno delle parentesi tonde che si trovano alla destra dell'identificatore nella dichiarazione di un metodo, sono dette **parametri** o **argomenti** del metodo.

Per esempio, nella seguente dichiarazione del metodo `somma()` vengono dichiarati due parametri interi `x` e `y`:

```
public int somma(int x, int y)
{
        return (x + y);
}
```

I parametri di un metodo saranno inizializzati, come abbiamo già potuto notare, al momento della chiamata del metodo. Infatti, per chiamare il metodo `somma()` dovremo passare i valori ai parametri, per esempio:

```
int risultato = oggettoAritmetica.somma(5, 6);
```

In particolare, durante l'esecuzione del programma, all'interno del metodo `somma()`, la variabile `x` varrà `5`, mentre `y` varrà `6`. Potremmo anche scrivere equivalentemente:

```
int a = 5, b = 6;
int risultato = oggettoAritmetica.somma(a, b);
```

ed ottenere lo stesso risultato. Infatti il passaggio di parametri avviene sempre per valore, il che significa che non vengono passate le variabili `a` e `b` al metodo `somma()`, bensì i loro valori, che all'interno del metodo saranno immagazzinati nei parametri `x` e `y`.

È importante sottolineare che un parametro si può considerare anche una variabile locale del metodo, essendo visibile solo all'interno del metodo e condividendo con esso il ciclo di vita. La differenza principale sussiste solo nella posizione della dichiarazione di una variabile locale (che potrebbe essere dichiarata alla fine del metodo) e quella di un parametro (che viene dichiarato contestualmente al metodo stesso), non nel processo di immagazzinamento in memoria.

Inoltre condividendo il ciclo di vita con il metodo in cui sono dichiarate, non ha senso (e non è possibile) anteporre alle variabili locali un modificatore d'accesso come `public`.

Il concetto e la modalità di allocazione di memoria di una variabile d'istanza si differenziano a tal punto dai concetti e dalle modalità di allocazione di memoria di una variabile locale (o di un parametro), che è possibile in una classe dare lo stesso nome ad una variabile locale (o ad un parametro) e ad una variabile d'istanza. Come vedremo questa pratica è molto utilizzata.

2.5 I metodi costruttori

In Java esistono metodi speciali che hanno delle proprietà particolari. Tra questi c'è da considerare il metodo costruttore, che possiede le seguenti caratteristiche:

1. ha lo stesso nome della classe;
2. non ha tipo di ritorno;
3. è chiamato automaticamente (e solamente) ogni volta che è istanziato un oggetto, relativamente a quell'oggetto;
4. è presente in ogni classe.
5. Solitamente un metodo costruttore viene definito allo scopo di inizializzare le variabili d'istanza.

2.5.1 Caratteristiche di un costruttore

Un costruttore ha sempre e comunque lo stesso nome della classe in cui è dichiarato. Ricordiamoci che è importante anche fare attenzione a lettere maiuscole e minuscole. Il fatto che non abbia tipo di ritorno non significa che il tipo di ritorno è `void` ma che non dichiara alcun tipo di ritorno! Si distingue dai metodi ordinari anche per questo. Per esempio, viene presentata una classe con un costruttore dichiarato esplicitamente:

```
public class Punto
{
    //metodo costruttore
    public Punto()
    {
        System.out.println("Costruito un Punto!");
    }
    public int x;
    public int y;
}
```

Si noti che verrà eseguito il blocco di codice del costruttore ogni volta che sarà istanziato un oggetto. Analizziamo meglio la sintassi che permette di istanziare oggetti. Per esempio:

```
Punto punto1; //dichiarazione
punto1 = new Punto();  //  istanza
```

che è equivalente a:

```
Punto punto1 = new Punto(); //dichiarazione ed istanza
```

Come accennato in precedenza, è la parola chiave `new` che istanzia formalmente l'oggetto. Perciò basterebbe la sintassi:

```
new Punto();
```

per istanziare l'oggetto. In questo modo, però, l'oggetto appena creato non avrebbe un "nome" (si dirà **reference**) e quindi non sarebbe utilizzabile. Solitamente quindi, quando si istanzia un oggetto, gli si assegna un reference dichiarato precedentemente. Appare quindi evidente che l'ultima parte dell'istruzione da analizzare (`Punto()`) non va interpretata come "nomeDellaClasse con parentesi tonde", bensì come "chiamata al metodo costruttore". In corrispondenza dell'istruzione suddetta, un programma produrrebbe il seguente output:

```
Costruito un Punto!
```

Questo è l'unico modo per chiamare un costruttore. Un costruttore essendo definito senza un tipo di ritorno non può considerarsi un metodo ordinario, e non si può chiamare in altri modi.

Nell'esempio precedente la chiamata al costruttore ha provocato la stampa a video di una frase. In realtà il costruttore solitamente è utilizzato per inizializzare le variabili d'istanza di un oggetto. Essendo un metodo (anche se speciale) può avere una lista di parametri. È quindi possibile codificare il seguente costruttore all'interno della classe `Punto`:

```
public class Punto
{
    public Punto(int a, int b)
    {
        x = a;
        y = b;
    }
    public int x;
    public int y;
}
```

Con questa classe non sarà più possibile istanziare gli oggetti con la solita sintassi:

```
Punto punto1 = new Punto();
```

otterremmo un errore di compilazione, dal momento che staremmo cercando di chiamare un costruttore che non esiste (quello senza parametri).

La sintassi da utilizzare per istanziare un oggetto invece, deve essere per esempio la seguente:

```
Punto punto1 = new Punto(5,6);
```

In realtà è anche possibile fornire più costruttori all'interno della stessa classe, basta che la lista degli argomenti in input differisca. Per esempio sarà possibile fornire entrambi i costruttori che abbiamo visto alla classe `Punto`:

```
public class Punto
{
    public Punto()
    {
        // Questo costruttore non fa niente
    }

    public Punto(int a, int b)
    {
        x = a;
        y = b;
    }
    public int x;
    public int y;
}
```

In questo modo potremo istanziare oggetti dalla classe `Punto` in due modi, o usando il primo costruttore:

```
Punto p = new Punto();
```

oppure usando il secondo:

```
Punto p = new Punto(12,14);
```

Nel primo caso il punto `p` avrà entrambe le coordinate `x` e `y` che valgono `0`. Ricordiamo infatti che le variabili d'istanza vengono inizializzate automaticamente ai valori nulli. Nel secondo caso `x` varrà `12` e `y` varrà `14`.

Notare che la seguente riga di codice:

```
Punto p = new Punto(12,14);
```

è equivalente alle seguenti:

```
Punto punto1 = new Punto();
punto1.x = 12;
punto1.y = 14;
```

La possibilità di creare più costruttori nella stessa classe è una manifestazione del polimorfismo (trattato in dettaglio nel capitolo 8) che si chiama overload. Questo consente di scrivere più metodi con lo stesso nome (ma con differente lista dei parametri). Ci sono altre regole da studiare ma le approfondiremo a tempo debito.

2.5.2 Costruttore di default

Quando creiamo un oggetto utilizzando la parola chiave `new`, c'è sempre una chiamata ad un costruttore (per esempio `new Punto()`). Eppure prima del paragrafo precedente non abbiamo mai fornito costruttori alle nostre classi, ma come appena detto, abbiamo chiamato costruttori ogni volta che abbiamo istanziato oggetti! Java infatti possiede una caratteristica molto importante che molti ignorano. Spesso il compilatore inserisce automaticamente e in modo trasparente istruzioni non inserite dal programmatore. Se infatti proviamo a compilare una classe sprovvista di costruttore, il compilatore ne fornirà uno implicitamente. Il costruttore inserito non contiene comandi che provocano qualche conseguenza visibile al programmatore. Esso è detto "costruttore di default" e non ha parametri. Ciò giustifica il fatto che fino ad ora non abbiamo mai istanziato oggetti passando parametri al costruttore. Se per esempio codifichiamo la classe `Punto` nel modo seguente:

```
public class Punto
{
    public int x;
    public int y;
}
```

Al momento della compilazione, il compilatore aggiungerebbe ad essa il costruttore di default:

```
public class Punto
{
    public Punto()
    {
       //nel costruttore di default
       //sembra non ci siano istruzioni...
    }
    public int x;
    public int y;
}
```

Ecco perché fino ad ora abbiamo istanziato gli oggetti di tipo `Punto` con la sintassi:

```
Punto p = new Punto();
```

Anche senza la presenza esplicita di costruttori. Se non esistesse il costruttore di default, avremmo dovuto imparare prima i costruttori e poi gli oggetti.

> **Questa è una delle caratteristiche che faceva sì che Java all'inizio fosse definito un linguaggio semplice da imparare. Infatti, il fatto che sia implicitamente inserito dal compilatore Java un costruttore all'interno delle classi, ci ha permesso di parlare di istanze di oggetti senza per forza dover prima spiegare un concetto come il costruttore.**

Sottolineiamo una volta di più che il costruttore di default viene inserito in una classe dal compilatore se e solo se il programmatore non ne ha fornito uno esplicitamente. Nel momento in cui il programmatore fornisce ad una classe un costruttore, sia esso con o senza parametri, il compilatore non inserirà quello di default. L'argomento sarà ulteriormente approfondito nel capitolo 7 dove scopriremo altre "istruzioni fantasma".

> **Un costruttore senza parametri inserito dal programmatore non si chiama costruttore di default, è solo un costruttore senza parametri! La terminologia è importante, non dobbiamo confondere due concetti diversi tra loro.**

L’ordine dei membri all’interno della classe non è importante al fine della compilazione. Possiamo scrivere prima i metodi, poi i costruttori e poi le variabili d’istanza, o alternare un costruttore a una variabile e ad un metodo; è tutto legale. La creazione di un oggetto di una classe in memoria, infatti, non provoca l’esecuzione in sequenza del codice della classe come se fosse un programma funzionale. Solitamente però, la convenzione richiede di definire prima le variabili d’istanza, poi i costruttori ed infine i metodi.

2.6 Introduzione ai package

Un `package` in Java permette di raggruppare in un’unica entità complessa classi Java logicamente correlate. Fisicamente il package non è altro che una **cartella** (sinonimo di **directory**) del nostro sistema operativo, ma non tutte le cartelle sono package. Per eleggere una cartella a package, una classe Java deve dichiarare nel suo codice la sua appartenenza a quel determinato package. Inoltre il risultato della compilazione della classe deve risiedere fisicamente all’interno di essa.

2.6.1 Package e import

Se volessimo far appartenere la classe `Punto` a un package `geometria`, dovremmo semplicemente inserire come prima istruzione del nostro file sorgente la seguente dichiarazione di package:

```
package geometria;

public class Punto
{
    //codice omesso
}
```

Questo comporterebbe che il file compilato (.class) dovrà risiedere in una cartella chiamata proprio `geometria`.

Se invece volessimo creare un albero di package più strutturato possiamo tranquillamente usare la sintassi sfruttando l’operatore dot (che anche in questo caso significa *appartenenza):*

```
package studio.matematica.geometria;

public class Punto
{
    //codice omesso
}
```

In questo caso il file compilato risiederà all'interno di una cartella `geometria` a sua volta contenuta in una cartella `matematica` a sua volta contenuta in una cartella `studio`.

Bisogna però tener conto delle regole di visibilità delle classi contenute nei package. Per esempio come fa una classe che si trova in una certa cartella ad utilizzare una classe che si trova in un'altra cartella? Se una classe chiamata `TestPunto` vuole usare la classe `Punto` e non si trova nello stesso package `studio.matematica.geometria`, dovrà importare la classe `Punto` con il comando `import`, che va dichiarato dopo la dichiarazione di package, ma prima della dichiarazione della classe, come nel seguente codice:

```
package studio.matematica.geometria.test;
import studio.matematica.geometria.Punto;

public class TestPunto
{
    public static void main (String args[])
    {
        Punto p = new Punto();
        . . .
    }
}
```

Supponiamo che i file sorgente `TestPunto.java` e `Punto.java` si trovino inizialmente nella stessa cartella `test`. Una volta compilati, verranno create le cartelle relative ai package dichiarati, e i file `.class` saranno direttamente spostati all'interno di esse come mostrato nella figura 2.1.

2.6.2 Gestione dei package manuale

Purtroppo però la situazione non è così semplice come sembra, infatti l'esempio precedente riporta uno degli scenari più semplici. Con EJE, questo tipo di

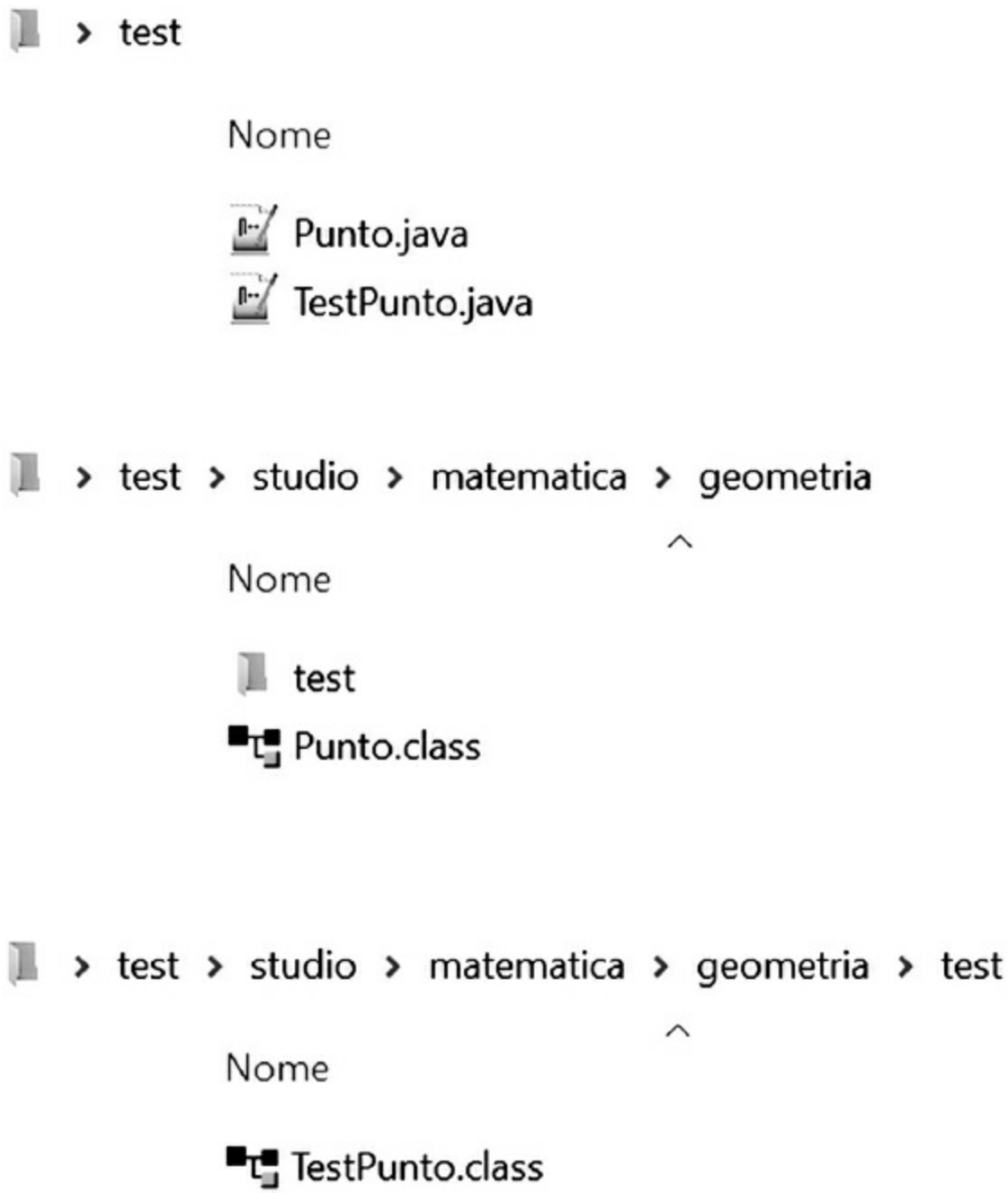


Figura 2.1 - Compilazione di file dichiarati con package.

scenario sarebbe coperto senza complicazioni, in quanto EJE gestirebbe per voi il posizionamento dei file e la creazione delle cartelle (a patto che i file `.java` si trovino entrambi nella stessa cartella iniziale). Se invece provassimo a compilare con la prompt dei comandi, dovremmo utilizzare la seguente sintassi per eseguire la compilazione, dopo esserci posizionati nella cartella `test`:

```
javac -d . Punto.java TestPunto.java
```

In questo modo compileremmo entrambi i file contemporaneamente specificando di voler creare le directory (sinonimo di cartelle) con l'opzione `-d` (abbreviativo di `-directory`) e specificando che tali cartelle devono essere create a partire dalla cartella corrente (che nell'informatica viene individuata solitamente con il simbolo di ".").

Per eseguire il file `TestPunto` invece, sempre dalla cartella `test` dovremmo lanciare il comando:

```
java studio.matematica.geometria.test.TestPunto
```

ovvero dovrete anteporre al nome del file `TestPunto`, la dichiarazione del suo package per indicare dove si trova il bytecode alla Java Virtual Machine.

Altri scenari che prevedono il posizionamento dei file sorgente in cartelle diverse richiedono ulteriori accorgimenti (spiegati in dettaglio nel capitolo 6) che potrebbero far diventare l'apprendimento di Java un vero e proprio incubo per il neofita. La soluzione è offerta dagli IDE, che gestiscono automaticamente il posizionamento dei file sorgente e dei file .class, alleggerendo così i compiti del programmatore.

Giacché il concetto di package non è essenziale al fine di approcciare correttamente al linguaggio, è stato scelto di rimandare al capitolo 6 la trattazione dettagliata di questo argomento. Fino a quel momento non dichiareremo più i package nei nostri esempi e nei nostri esercizi.

2.7 Altri componenti fondamentali della programmazione Java

L'approccio che seguiremo è quello di presentare gli argomenti nel momento opportuno senza forzare il lettore ad imparare concetti per i quali non è pronto. Per completezza però in questo paragrafo vogliamo solo introdurre altri componenti fondamentali che studieremo approfonditamente più avanti.

2.7.1 Interfacce

Un**'interfaccia** può essere salvata e compilata all'interno di un file con

suffisso .java che abbia lo stesso nome dell'interfaccia, esattamente come una classe. Un'interfaccia però non è istanziabile: non è possibile creare oggetti a partire da un'interfaccia. In essa è possibile definire solamente determinati tipologie di metodi (solitamente metodi astratti, ovvero senza blocco di codice e quindi non invocabili direttamente) e non si possono dichiarare variabili d'istanza ma solo costanti.

Un esempio di interfaccia potrebbe essere la seguente:

```
public interface Suonabile
{
    public void suona(String nota);
}
```

Come è possibile osservare la sintassi è molto simile a quella della classe, ma si usa la parola chiave `interface` in luogo della parola chiave `class`. Possiamo notare inoltre che il metodo `suona()` dichiarato all'interno dell'interfaccia, non ha un blocco di codice ma termina con un `;` (si dice **metodo astratto).** Questo implica che non è possibile invocare un metodo di un'interfaccia in quanto esso non è definito. Le interfacce sono strettamente legate al concetto di **ereditarietà**, un paradigma della programmazione object oriented che permette di mettere in relazione di estensione più classi. Le interfacce saranno estese da classi, e queste implementeranno i metodi astratti che verranno ereditati dalle interfacce. Per esempio la classe:

```
public class Ukulele implements Suonabile
{
    public void suona(String nota)
    {
        System.out.println("L'ukulele sta suonando la nota "
+ nota);
    }
}
```

utilizza la parola chiave implements per estendere ed **implementare** i metodi che vengono ereditati dall'interfaccia. Ma anche la seguente classe implementa l'interfaccia `Suonabile`:

```
public class Armonica implements Suonabile
{
    public void suona(String nota)
```

```
    {
        System.out.println("L'armonica sta suonando la nota
" + nota);
    }
}
```

Nel metodo `main()` della seguente classe, potremmo invocare il metodo `suona()` definito nell'interfaccia `Suonabile`, su oggetti istanziati dalle due classi `Ukulele` e `Armonica`:

```
public class TestSuonabili
{

    public static void main(String args[])
    {
        Armonica armonica = new Armonica();
        Ukulele ukulele = new Ukulele();
        armonica.suona("DO");
        ukulele.suona("RE");
    }
}
```

Le due classi implementano lo stesso metodo `suona()` in modo diverso, realizzando un altro paradigma della programmazione ad oggetti chiamato **polimorfismo** (cfr. capitolo 8). Le interfacce e l'ereditarietà saranno trattate dettagliatamente nel capitolo 7, mentre il polimorfismo nel capitolo 8.

2.7.2 Enumerazioni

Un'**enumerazione** può essere salvata e compilata all'interno di un file con suffisso `.java` che abbia lo stesso nome dell'enumerazione, esattamente come una classe o un'interfaccia. Le enumerazioni definiscono tutte le loro istanze al loro interno, non è possibile istanziare altri oggetti di un'enumerazione all'esterno dell'enumerazione stessa. Hanno proprio lo scopo di definire un tipo che può essere istanziato solo con un numero limitato di oggetti, che è un'esigenza tipica nella programmazione. Per esempio, se avessimo bisogno di un oggetto che rappresenta un giorno della settimana, creare una classe sarebbe dispendioso e per certi versi inutile. Infatti dovremmo implementare i controlli affinché gli oggetti da istanziare da questa classe siano solo sette. Invece con una enumerazione possiamo

scrivere direttamente:

```
public enum GiornoDellaSettimana
{
    LUNEDI, MARTEDI, MERCOLEDI, GIOVEDI, VENERDI, SABATO,
DOMENICA;
}
```

Tutte le potenzialità delle enumerazioni saranno dettagliate nel capitolo 10.

2.7.3 Annotazioni

Anche un**'annotazione** può essere salvata e compilata all'interno di un file con suffisso `.java` che abbia lo stesso nome dell'annotazione, esattamente come una classe, un'interfaccia o un'enumerazione. Il concetto di annotazione è abbastanza complesso, e molti programmatori Java, più che creare annotazioni, usano le annotazioni definite nella libreria standard e in altre librerie. Le annotazioni sono metadati con cui possiamo caratterizzare altri elementi come classi, metodi, variabili, interfacce etc. Le annotazioni vengono lette ed interpretate da altri software (per esempio il compilatore, la Java Virtual Machine o un nostro programma) che possono leggere gli elementi di codice annotati. Per esempio l'annotazione della libreria standard `@FunctionalInterface`, può annotare interfacce che dichiarano un unico metodo. Essa ha proprio il compito di dichiarare al compilatore che l'interfaccia annotata deve dichiarare un unico metodo. Nel caso l'interfaccia annotata non rispetti questo vincolo, il compilatore segnalerà un errore al programmatore. Per esempio se compiliamo la seguente interfaccia:

```
@FunctionalInterface
public interface InterfacciaConDueMetodi
{
    void metodo1();
    void metodo2();
}
```

il compilatore restituirà il seguente errore:

```
InterfacciaConDueMetodi.java:1: error: Unexpected
@FunctionalInterface annotation
@FunctionalInterface
```

```
^
  InterfacciaConDueMetodi is not a functional interface
    multiple non-overriding abstract methods found in
interface
InterfacciaConDueMetodi
1 error
```

Possiamo anche creare noi le nostre annotazioni con la seguente sintassi:

```
public @interface MiaAnnotazione
{
    //...
}
```

Alle annotazioni è dedicato l'intero capitolo 14.

2.7.4 Moduli

I **moduli** sono la novità più importante di Java 9. Ci consentiranno di gestire la struttura delle nostre applicazioni in maniera più pulita e funzionale. Come i package i moduli sono fisicamente delle cartelle. Queste possono contenere package, classi ed altre risorse e un **file descrittore.** Infatti all'interno della cartella che deve rappresentare il modulo ci deve essere un file `.java` chiamato per convenzione `module-info.java` che dichiara il nome e le caratteristiche del modulo. Per esempio con la seguente sintassi:

```
module com.cdsc.test
{
    exports com.cdsc.test;
}
```

stiamo dichiarando un modulo di nome `com.cdsc.test`. Esso dichiara che siano visibili al suo esterno tutte le classi appartenenti al package `com.cdsc.test`.

I moduli sono utili soprattutto per applicazioni molto grandi, che hanno bisogno di essere organizzate al meglio. Per tale ragione saranno trattati dettagliatamente solo alla fine di questo libro.

2.7.5 Inizializzatori

Gli **inizializzatori** sono componenti poco utilizzati e vanno dichiarati internamente alle classi. Come dice il nome gli inizializzatori sono usati per inizializzare la classe o l'oggetto. Infatti esistono **inizializzatori di classe** (detti anche **inizializzatori statici)** e **inizializzatori di istanza.** Hanno la caratteristica di essere automaticamente invocati una sola volta nel momento in cui viene caricata la classe (inizializzatori statici) o l'oggetto (inizializzatori d'istanza). Per esempio nella classe:

```
public class TestInizializzatore
{
    {
        System.out.println("Questo è un inizializzatore
d'istanza");
    }
}
```

abbiamo dichiarato un inizializzatore d'istanza. La sintassi è semplicissima (due parentesi graffe) e questo verrà chiamato ogni volta che verrà caricato in memoria un oggetto istanziato dalla classe `TestInizializzatore`, persino prima di un eventuale costruttore. Gli inizializzatori verranno trattati nel capitolo 6.

2.7.6 Classi innestate

Le **classi innestate** sono classi definite all'interno di altre classi. Godono di alcune proprietà tra le quali quella di avere accesso a tutte le variabili definite all'interno della classe le contengono. Segue un esempio:

```
public class ClasseEsterna
{
    private int i = 0;
    class ClasseInterna
    {
        public void metodo()
        {
            System.out.println("Posso accedere alla
variabile i " + i);
        }
    }
}
```

Esistono anche **classi innestate anonime** (senza nome) che si possono dichiarare al volo in maniera locale all'interno di un metodo.

2.7.7 Espressioni lambda

Le **espressioni lambda** sono state introdotte solo con la versione 8 di Java, rivoluzionando completamente il linguaggio. Con le espressioni lambda possiamo assegnare un reference ad un metodo e passarlo in input ad un altro metodo come se questo fosse una variabile. Si tratta di un argomento complesso che verrà trattato nel capitolo 16. La sintassi è abbastanza particolare. Un'espressione lambda ci permette di definire al volo il codice del suo unico metodo e passarlo in input ad un metodo. Questo potrà eseguire il codice dell'espressione lambda in qualsiasi momento all'interno di un flusso di codice ben definito. Per esempio, se consideriamo l'interfaccia funzionale `Consumer`, con il seguente codice:

```
Consumer consumer = ((s)->System.out.println(s));
```

abbiamo dichiarato un reference ad un'espressione lambda (una funzione) che può essere utilizzato come fosse una variabile ed eseguito quando si vuole.

Riepilogo

Lo **sviluppo del software** richiede che siano portate a termine una serie di attività (un **processo)** che permetteranno ad un'idea di diventare software. Un processo minimale richiede almeno quattro attività ben definite: capire cosa fare, capire come fare, implementare il software, testarlo e risolvere gli eventuali problemi.

Java è un **linguaggio object oriented**, il che significa che supporta con le sue caratteristiche la programmazione object oriented **(OOP).** Alla base della programmazione OOP ci sono i concetti di classe ed oggetto, che come tutti gli altri concetti sono ispirati al mondo reale.

Una **classe** rappresenta un'astrazione per un gruppo di oggetti che condividono le stesse caratteristiche e funzionalità. Un **oggetto** è una creazione fisica (detta anche **istanza)** della classe. Ogni classe è unica, ma gli oggetti creati a partire dalla stessa classe sono diversi.

I programmi Java saranno costituiti da tante classi che si referenziano l'un l'altra. Nei programmi Java però, saranno gli oggetti istanziati dalle classi che eseguiranno le istruzioni, le classi invece serviranno soprattutto per definire la struttura degli oggetti. Un'eccezione è rappresentata dalla classe che contiene il metodo statico `main()`, che è il punto di partenza per ogni programma Java e di conseguenza esegue delle istruzioni (di solito crea oggetti che eseguono istruzioni). La classe contenente il metodo `main()` è l'unica classe che si può eseguire mediante il comando `java`.

Ogni file che fa parte di un'applicazione Java conterrà il listato di una ed una sola classe (tranne in alcuni casi eccezionali che per ora il lettore può ignorare). Inoltre, ogni classe solitamente contiene definizioni di variabili, metodi e costruttori.

I **metodi** contengono codice operativo e definiscono le funzionalità degli oggetti. Hanno bisogno di essere dichiarati all'interno di una classe e di essere utilizzati tramite l'operatore **dot** applicato ad un oggetto. I metodi possono o meno restituire un risultato in output ed avere o meno dei parametri in input.

I **varargs** sono un particolare parametro che permette di passare un numero variabile di valori ad un metodo. È possibile utilizzare un unico varargs come parametro di un metodo, e in presenza di altri parametri deve essere posizionato come ultimo della lista.

Le **variabili** si dividono in tre tipologie, definite dal posizionamento della dichiarazione: d'istanza, locali e parametri (che si possono in sostanza considerare locali). Una **variabile d'istanza** è dichiarata al di fuori di un metodo e rappresenta un membro della classe al pari di un metodo. Oltre ad un nome (identificatore), le variabili hanno un tipo (per esempio `int`) e un valore (per esempio `8`). In particolare le variabili d'istanza rappresentano gli attributi (le caratteristiche) di un oggetto ed è possibile accedervi tramite l'operatore dot applicato ad un oggetto. Le **variabili locali** ed i **parametri** di un metodo sono invece dichiarate all'interno di un metodo con cui condividono il ciclo di vita. Questo significa che in fase di esecuzione del programma, vengono create quando viene eseguito il metodo in cui sono dichiarate, e una volta terminato il metodo non saranno più accessibili. I parametri in particolare vengono inizializzati con valori che provengono dall'esterno del metodo in cui sono definiti, e tali valori rappresentano l'input

di un metodo.

I **costruttori** sono metodi speciali che si trovano all'interno di ogni classe. Infatti, se il programmatore non ne fornisce uno esplicitamente, il compilatore Java aggiungerà il **costruttore di default.** Il costruttore non ha tipo di ritorno, ha lo stesso nome della classe in cui è dichiarato e viene chiamato ogni volta che viene istanziato un oggetto. Può prendere in input parametri come un metodo qualsiasi e grazie ad una caratteristica chiamata overload possiamo definire più costruttori all'interno della stessa classe. Un costruttore di solito viene utilizzato per inizializzare le variabili d'istanza di un oggetto.

Un **package** consente di raggruppare le classi logicamente relazionate tra loro. Un package fisicamente coincide con una cartella. Una classe che vuole utilizzarne un'altra appartenente da un package diverso, la deve **importare** mediante il comando `import`. Durante lo sviluppo, la gestione dei package può risultare piuttosto macchinosa se non si usa un IDE.

Altri componenti fondamentali sono stati solo accennati. Un**'interfaccia** solitamente contiene metodi astratti che vengono implementati all'interno di altre classi. Le **enumerazioni** rispondono all'esigenza di avere un numero finito di oggetti o valori. Le **annotazioni** invece vengono utilizzate per annotare il nostro codice, rendendolo analizzabile da altri software come ad esempio il compilatore. I **moduli** invece sono cartelle che si auto-descrivono tramite un file java, e permettono di incapsulare package, classi e altre risorse, ed esporle ad altri moduli all'occorrenza. Gli **inizializzatori** sono poco utilizzati, e vengono eseguiti quando una classe o un oggetto è caricato in memoria. Le **classi innestate** sono classi dichiarate all'interno di altre classi che possono accedere a tutti i membri della classe in cui sono dichiarate. Infine, le **espressioni lambda** permettono di utilizzare le funzioni con un reference come fossero variabili.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere cosa significa sviluppare un programma in Java, cosa sono i processi e i paradigmi di programmazione (unità 2.1)	☐	
Saper definire i concetti di classe, oggetto, variabile, metodo e costruttore (unità 2.2, 2.3, 2.4, 2.5)	☐	
Saper dichiarare una classe (unità 2.2)	☐	
Istanziare oggetti da una classe (unità 2.2)	☐	
Utilizzare i membri pubblici di un oggetto sfruttando l'operatore dot (unità 2.2, 2.3, 2.4)	☐	
Dichiarare ed invocare un metodo (unità 2.3)	☐	
Saper dichiarare e assegnare un valore ad una variabile (unità 2.4)	☐	
Saper definire ed utilizzare i diversi tipi di variabili (d'istanza, locali e parametri formali) (unità 2.4)	☐	
Dichiarare ed invocare un metodo costruttore (unità 2.5)	☐	
Saper definire un costruttore di default (unità 2.5)	☐	
Saper definire una classe appartenente ad un package e saper importare classi da altri package (unità 2.6)	☐	
Avere un'idea di tutti gli altri componenti fondamentali della programmazione Java che studieremo nei prossimi capitoli (unità 2.7)	☐	

Note:

3

Stile di codifica, tipi di dati ed array

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Conoscere le regole e le convenzioni per dichiarare identificatori per il codice Java (unità 3.1).
- ✔ Comprendere come viene gestita la memoria, i sistemi numerici posizionali e il metodo del complemento a due (unità 3.2).
- ✔ Conoscere e saper utilizzare tutti i tipi di dati primitivi (unità 3.3).
- ✔ Saper gestire casting e promotion (unità 3.3).
- ✔ Saper utilizzare i reference e capirne la filosofia (unità 3.4).
- ✔ Iniziare ad esplorare la documentazione della libreria standard di Java (unità 3.5).
- ✔ Saper utilizzare la classe String (unità 3.5).
- ✔ Saper commentare il proprio codice ed essere in grado di utilizzare lo strumento `javadoc` per produrre documentazione tecnica esterna (unità 3.1, 3.5).
- ✔ Saper definire, creare, ed inizializzare gli array (unità 3.6).

In questo capitolo saranno dapprima spiegate alcune regole che la codifica Java richiede. Queste regole sono molto semplici ma fondamentali per

sviluppare. Utilizzare uno stile standard semplifica la vita al programmatore e renderà più leggibile il suo codice. Verranno anche introdotti alcuni concetti di base sulla gestione della memoria, nozioni fondamentali per il neofita.

Successivamente saranno presentati i tipi di dati definiti dal linguaggio e le relative problematiche. Sino ad ora abbiamo quasi sempre utilizzato il tipo di dato intero `int`, dopo lo studio di questo capitolo potremo finalmente padroneggiare tutti i tipi di dato che definisce Java. Inoltre faremo la conoscenza del fondamentale concetto di reference. Sarà inoltre introdotta anche la libreria standard di Java, che ci permetterà di utilizzare oggetti di alto livello e sfruttarli come tipi di dati complessi all'interno dei nostri programmi. Per completare il discorso sarà presentato ed approfondito il discorso sugli array, un argomento che risulta piuttosto ostico a molti programmatori.

Un array rappresenta un tipo di dato complesso che definisce una collezione di elementi indicizzati di un certo tipo. Con un solo nome potremo accedere agli elementi della collezione mediante un indice.

3.1 Stile di codifica

Con il termine **stile di codifica** intendiamo una serie di regole obbligatorie e non, che caratterizzano solitamente il codice Java. Delle regole obbligatorie non si può fare ovviamente a meno, ma anche le facoltative sono molto importanti, perché utilizzate dai programmatori di tutto il mondo, e quindi il loro uso semplifica enormemente la leggibilità del codice.

Il linguaggio Java:

- è a schema libero.
- Supporta vari tipi di commenti.
- Definisce parole chiave.
- È case sensitive.
- Ha regole ferree per i tipi di dati ed alcune semplici convenzioni per i

nomi.

3.1.1 Schema Libero

Java è un linguaggio a **schema libero.** Questo non significa che possiamo violare le regole del linguaggio. Per esempio l'istruzione di dichiarazione di package deve essere sempre la prima in un file, le istruzioni di importazione devono seguire l'eventuale dichiarazione di package, la dichiarazione di una classe non può precedere le dichiarazioni di `package` e di `import`, le classi e i metodi sono delimitate da parentesi graffe e così via. Queste ed altre regole non possono essere violate perché il compilatore ci restituirebbe errori in compilazione. Java è un linguaggio a schema libero nel senso che potremmo scrivere un intero programma in Java tutto su di un'unica riga, oppure andando a capo dopo ogni parola: il compilatore non avrebbe problemi a compilare il codice se esso è corretto. Il problema semmai sarebbe dello sviluppatore che avrebbe grandi difficoltà a comprendere il codice!

Esistono quindi metodi standard d**'indentazione del codice** che facilitano la lettura di un programma Java. Di seguito è presentata una semplice classe che utilizza uno dei due più usati metodi di formattazione:

```
public class Classe
{
    public int intero;
    public void metodo()
    {
        intero = 29;
        int unAltroIntero = 7;
        System.out.println(intero + altroIntero);
    }
}
```

Con questo stile (che è utilizzato anche dai programmatori del linguaggio C) il lettore può capire subito dove la classe ha inizio e dove ha fine, dato che le parentesi graffe che delimitano un blocco di codice sono incolonnate. Stesso discorso per il metodo: è chiaro dove inizia, dove finisce ed il codice che ne definisce il comportamento. È lo stile che abbiamo utilizzato fino a questo punto, perché ci è sembrato didatticamente più valido per i neofiti.

Ma lo stile più utilizzato dai programmatori Java è il seguente:

```
public class Classe {
    public int intero;
    public void metodo() {
        intero = 29;
        int unAltroIntero = 7;
        System.out.println(intero + altroIntero);
    }
}
```

per il quale valgono le stesse osservazioni fatte per il primo metodo.

Si raccomanda, per una buona riuscita del lavoro che sarà svolto in seguito dal lettore, una rigorosa applicazione di uno dei due stili appena presentati. Ovviamente il consiglio è di utilizzare il secondo stile che abbiamo presentato, che per Java è uno standard. Pochissimi programmatori Java utilizzano il primo stile, e di solito si tratta di sviluppatori che programmavano in precedenza in C e che sono più restii a cambiare abitudine.

Se utilizzate EJE come editor potete formattare il vostro codice con entrambi gli stili automaticamente mediante il menu apposito, o tramite il relativo bottone sulla barra degli strumenti, o attraverso la scorciatoia di tastiera CTRL-MAIUSC-F. Per configurare lo stile da utilizzare, scegliere il menu File ⇨ Options (o premere F12) e impostare il Braces style con lo stile desiderato nel tab Editor.

Alcuni dei tipici errori che il programmatore alle prime armi commette sono semplici dimenticanze. È frequente dimenticare di chiudere una parentesi graffa di una classe, o dimenticare il “;” per terminare uno statement o anche le parentesi tonde che seguono l’invocazione di un metodo. Per questo è consigliato al lettore di abituarsi a scrivere ogni istruzione in maniera completa, per poi pensare a formattare in maniera corretta. Per esempio, se dichiariamo una classe è buona prassi scrivere entrambe le parentesi graffe, prima di scrivere il codice contenuto in esse. I tre passi seguenti dovrebbero chiarire meglio il concetto:

- passo 1: dichiarazione:

```
public class Classe {}
```

- passo 2: formattazione:

```
public class Classe {

}
```

- passo 3: completamento:

```
public class Classe {
    public int intero;
    . . .
}
```

3.1.2 Commenti

Commentare opportunamente il codice è una pratica che dovrebbe essere considerata obbligatoria dal programmatore che inizia a programmare. I commenti non solo chiariscono il codice ma obbligano anche a capire bene cosa si sta scrivendo visto che bisogna descriverlo. Quindi i commenti dovrebbero favorire molto l'apprendimento del neofita. Per i programmatori esperti invece, la maggior parte dei commenti dovrebbero diventare quasi inutili, perché il codice idealmente dovrebbe risultare talmente leggibile da rendere inutile la pratica di commentare il codice scritto. In effetti il codice non mente mai, i commenti a volte sì!

In qualsiasi caso, Java supporta tre modi diversi per commentare il codice:

1. **Commenti a singola riga** (letteralmente **single line comments**):

```
// Questo è un commento su una sola riga
```

2. **Commenti multilinea** (letteralmente **multiline comments**):

```
/*
   Questo è un commento
   su più righe
*/
```

3. **Commenti di documentazione** (letteralmente **documentation comments**), conosciuti anche come **commenti Javadoc**:

```
/**
   Questo commento permette di produrre
   la documentazione del codice
   in formato HTML, nello standard Javadoc
*/
```

Per i commenti a singola riga tutto ciò che scriveremo su di una riga che segue i simboli "//" non sarà preso in considerazione dal compilatore. Questa sintassi permetterà di commentare brevemente alcune parti di codice. Per esempio:

```
// Questo è un metodo
public void metodo() {
     ...
}

public int a; //Questa è una variabile
```

Questo tipo di commenti può risultare particolarmente utile ai neofiti per commentare con la giusta terminologia ogni istruzione che codificherà. È una pratica davvero fondamentale per chi inizia. Il commento su più righe potrebbe essere utile, ad esempio, per la descrizione delle funzionalità di un programma. La sintassi richiede di includere il testo da considerare come commento tra i simboli "`/*`" e "`*/`". Per esempio:

```
/*
    Questo metodo è commentato...
    public void metodo() {
         ...
    }
*/
```

Questo tipo di commento è spesso utilizzato per commentare parti di codice che non si è ancora sicuri di eliminare definitivamente, ma anche per fare commenti più lunghi su più linee.

Queste due prime tipologie di commenti sono importate dal

linguaggio C++.

Il terzo tipo di commento è del tutto simile al commento multilinea. A livello di sintassi l'unica differenza è che il commento si apre con i simboli "`/**`" (notare l'asterisco in più) ma si chiude allo stesso modo con i simboli "`*/`". Il commento di documentazione però offre in più la possibilità di produrre in modo standard, in formato HTML, la **documentazione tecnica** del programma, sfruttando un comando chiamato javadoc. Per "documentazione tecnica" intendiamo la documentazione che spiega il funzionamento dei componenti Java (principalmente i metodi e i costruttori, ma non solo) descrivendo i parametri di input, quelli di output, e il comportamento (ovvero ciò che fa) con una sintassi precisa e standard. Vedremo delle applicazioni di tale tipologia di commento più avanti nel capitolo, nel paragrafo intitolato "Lo strumento Javadoc".

3.1.3 Regole per gli identificatori

Gli identificatori (sinonimo di nomi) dei metodi, delle classi, degli oggetti, dei package, delle variabili e delle costanti (ma anche delle interfacce, delle enumerazioni e delle annotazioni che studieremo più avanti) hanno due regole da rispettare obbligatoriamente.

1. La prima regola da ricordare per creare un identificatore per un qualsiasi componente Java è che può usare solo un certo insieme di caratteri, numeri e simboli. In particolare, in un identificatore:
   - il primo carattere può essere `A-Z, a-z, _, $`
   - il secondo ed i successivi possono essere `A-Z, a-z, _, $, 0-9`

   Quindi: `a2` è un identificatore valido, mentre `2a` non lo è.
2. Per la seconda regola, un identificatore non può coincidere con una

parola chiave (keyword) di Java. Infatti una parola chiave ha un certo significato per il linguaggio di programmazione. Nella seguente tabella sono riportate tutte le parole chiave di Java in ordine alfabetico e che quindi non possono essere utilizzate come identificatori:

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	if	goto
implements	import	instanceof	int	interface	long
module	native	new	open	opens	package
private	protected	provides	public	requires	return
short	static	strictfp	super	switch	synchronized
this	throw	throws	to	transient	transitive
try	uses	void	volatile	while	with
@interface	_				

Quindi non potremo chiamare una variabile `class`, oppure un metodo `void`.

Bisogna anche ricordare che Java è un linguaggio **case sensitive**, ovvero fa distinzione tra lettere maiuscole e minuscole. Il programmatore alle prime armi tende a digerire poco questa caratteristica del linguaggio. Per esempio non è possibile dichiarare una classe scrivendo `class` con lettera iniziale maiuscola, perché per il compilatore non significa niente. L'identificatore `unAltroIntero` è diverso dall'identificatore `unaltroIntero`. Bisogna quindi fare attenzione e, specialmente nei primi tempi, avere un po' di pazienza.

Si noti che tutte le parole nella tabella precedente contengono solo lettere minuscole (oltre ai simboli _ e @).

Possiamo notare alcune parole chiave che abbiamo già incontrato come `package`, `new`, `static`, `import`, `int`, `public`, `void`, `return` e `class`, e di cui già conosciamo (o quasi) il significato. Bisogna però precisare che nella tabella riportata sopra non tutte le parole sono keyword.

Vengono chiamate **parole riservate (reserved word)** `goto`, `const` e `_` (quest'ultimo inteso come simbolo di sottolineatura, che in inglese solitamente chiamiamo **underscore**). Questi infatti non hanno un significato per il linguaggio, ma semplicemente non si possono utilizzare come identificatori. Le parole chiave `goto` e `const` sono parole riservate da quando è nato il linguaggio, mentre per quanto riguarda _ è una novità di Java 9. In particolare è previsto che questo simbolo venga riabilitato nella prossima versione di Java per diventare una parola chiave.

> **Notare anche che `@interface` (parola utilizzata per dichiarare le annotazioni, di cui parleremo più avanti) iniziando con il simbolo di "chiocciola" `@`, non potrebbe comunque essere utilizzata quale identificatore, come spiegato all'inizio di questo paragrafo. Anche in questo caso quindi non si dovrebbe parlare tecnicamente di parola chiave.**

Anche altre parole della precedente tabella sono state introdotte con Java 9: `exports`, `module`, `open`, `opens`, `provides`, `requires`, `to`, `transitive`, `uses` e `with`. Queste vengono dette **parole a utilizzo limitato (restricted word)**, in quanto non sono utilizzabili come identificatori solo nell'ambito della dichiarazione di un modulo (novità di Java 9). Quindi potremmo dire che sono keyword solo nel caso di dichiarazione di moduli.

3.1.4 Standard e convenzioni per i nomi

Se definiamo gli identificatori rispettando le due regole appena descritte non otterremo errori in compilazione. Ma esistono direttive standard fornite direttamente da Oracle (e originariamente da Sun Microsystems) per raggiungere uno standard anche nello stile d'implementazione. È importantissimo utilizzare queste direttive in un linguaggio tanto standardizzato come Java.

Gli identificatori devono essere significativi. Infatti, se scriviamo un programma utilizzando la classe `a`, che definisce le variabili `b`, `c`, `d` e il metodo `e`, sicuramente ridurremo la comprensibilità del programma stesso.

Di solito l'identificatore di una variabile è composto da uno o più sostantivi, per esempio `numeroLati`, o `larghezza` o anche `numeroPartecipantiAlSimposio`. Gli identificatori dei metodi solitamente conterranno verbi, ad esempio `stampaNumeroPartecipantiAlSimposio`, o `somma`.

Esistono convenzioni per gli identificatori a seconda del tipo di componente a cui bisogna dare un nome, sono semplicissime. Eccole elencate.

- **Convenzione per le classi**

  Un identificatore di una classe (ma questa regola vale anche per le interfacce, le enumerazioni e le annotazioni che studieremo più avanti) deve sempre iniziare con una lettera maiuscola. Se composto da più parole, queste non si possono separare, perché il compilatore non può intuire le nostre intenzioni. Come abbiamo notato in precedenza, bisogna invece unire le parole in modo tale da formare un unico identificatore, e fare iniziare ognuna di esse con lettera maiuscola. Esempi di identificatori per una classe potrebbero essere:

  - `Persona`
  - `MacchinaDaCorsa`
  - `FiguraGeometrica`

- **Convenzione per le variabili**

  Un identificatore di una variabile deve sempre iniziare con una lettera minuscola. Se l'identificatore di una variabile deve essere composto da più parole, valgono le stesse regole in uso per gli identificatori delle classi (tranne il fatto che la prima lettera deve sempre essere minuscola). Quindi esempi di identificatori per una variabile potrebbero essere:

  - `pesoSpecifico`
  - `numeroDiMinutiComplessivi`
  - `x`

- **Convenzione per i metodi**

  Per un identificatore di un metodo valgono le stesse regole in uso per gli identificatori delle variabili. Potremo in ogni caso sempre distinguere un identificatore di una variabile da un identificatore di un metodo, giacché quest'ultimo è sempre seguito da parentesi tonde. Inoltre, per quanto già affermato, il nome di un metodo dovrebbe contenere almeno un verbo. Quindi, esempi di identificatori validi per un metodo potrebbero essere:

  - `sommaDueNumeri(int a, int b)`
  - `cercaUnaParola(String parola)`
  - `stampa()`

- **Convenzione per le costanti**

  Gli identificatori delle costanti, invece, si devono distinguere nettamente dagli altri, e tutte le lettere dovranno essere maiuscole. Se l'identificatore è composto di più parole, queste si separano con un underscore (simbolo di sottolineatura). Per esempio:

  - `NUMERO_LATI_DI_UN_QUADRATO`
  - `PI_GRECO`

- **Convenzione per i package**

  I nomi dei package contengono generalmente solo lettere minuscole. Se abbiamo bisogno di parole composte è possibile creare package innestati. Per esempio se volessimo creare un package di assistenza clienti, è possibile creare due package innestati nel seguente modo:

  - `assistenza.clienti`

Non c'è una convenzione sul numero di caratteri massimo da usare per un identificatore, ma in Java è meglio preferire la chiarezza alla brevità di un nome. Ovviamente con un po' di buon senso.

3.2 Gestione della memoria

Nel primo capitolo abbiamo introdotto alcuni concetti base di informatica. Abbiamo asserito che esistono due tipologie di memoria: la **memoria principale** (che abbiamo associato alla memoria RAM) e la **memoria secondaria** (che abbiamo associato all'hard disk). La memoria secondaria è una memoria permanente (ovvero sopravvive allo spegnimento del computer). Questa è utilizzata per salvare i dati sotto forma di **file** e **cartelle.** La memoria RAM invece è volatile e ottimizzata per la scrittura e il recupero dei dati. Nella memoria RAM vengono caricati i programmi con i propri dati quando essi vengono eseguiti.

È il **processore** che gestisce la memoria, decidendo in che parte di essa devono essere memorizzati tali dati. Infatti possiamo immaginare la memoria come una lunga sequenza di celle che possono memorizzare uno solo tra due valori, che per convenzione assumiamo essere `0` e `1`. Queste celle vengono dette **bit**, abbreviativo di **binary digit**, ovvero "cifra binaria".

> **Ricordiamo che il sistema numerico binario (di cui parleremo tra poco) contempla solo i simboli `0` e `1`, che rappresentano anche l'intero vocabolario del linguaggio macchina.**

All'interno di queste celle vengono memorizzati i dati dal processore. Per fare questo, la memoria viene suddivisa in sequenze di otto bit che vengono chiamate **byte**, che hanno la caratteristica di avere un **indirizzo**, ovvero di poter essere individuate da un numero.

3.2.1 Byte

Quindi un **byte** è composto da otto bit e ad esso è associato un **indirizzo** numerico per essere individuato. **Otto bit** sono sufficienti per memorizzare ogni singolo simbolo presente su una tastiera. Infatti come vedremo nel prossimo paragrafo, tramite il sistema numerico binario con un byte possiamo rappresentare una certa quantità di numeri ed ogni numero può essere tradotto in uno qualsiasi dei simboli presente sulla tastiera mediante uno standard di

decodifica.

Il fatto che un byte abbia un **indirizzo** numerico, permette al processore di tenere traccia di dove i dati vengono immagazzinati, per poi recuperarli velocemente in futuro.

> **L'indirizzo viene scelto non dall'hardware ma dal programma. Nel caso di Java sarà la Java Virtual Machine ad occuparsi di tutto, il programmatore non interagirà mai con la memoria di basso livello.**

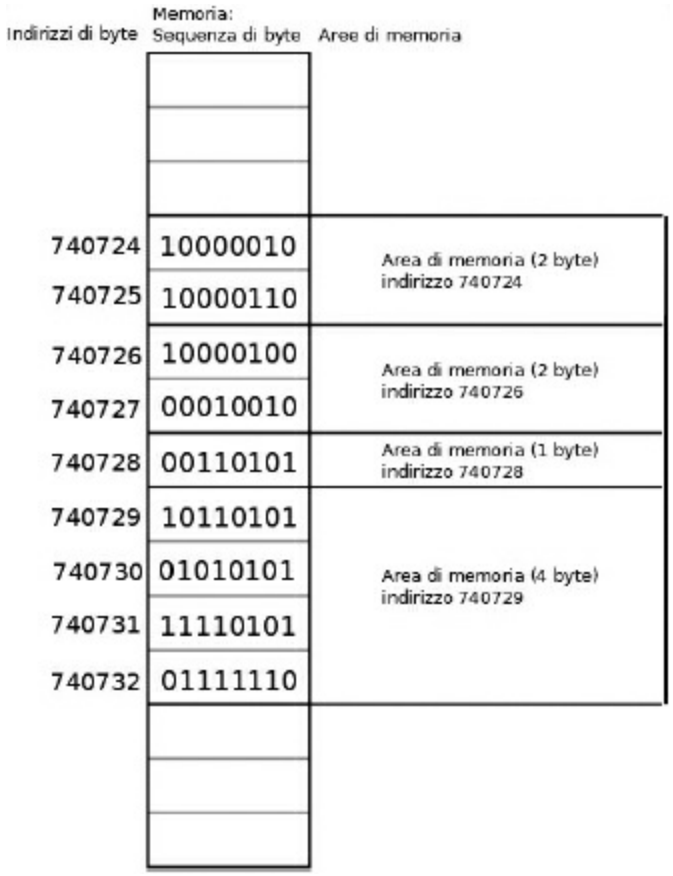


Figura 3.1 - Rappresentazione della memoria.

Se un dato è troppo grande per essere immagazzinato in un byte, allora esso verrà memorizzato anche su altri byte adiacenti, che verranno considerati un'unica **area di memoria** il cui indirizzo coinciderà con l'indirizzo del primo byte. Nella figura 3.1, viene rappresentato un esempio di come alcuni

dati potrebbero essere immagazzinati in memoria.

Le sequenze di 8 bit (i byte) rappresentano numeri. Per interpretarli bisogna utilizzare il **sistema numerico binario** e considerare che Java utilizza il metodo del **complemento a due** per la rappresentazione dei numeri all'interno di byte.

3.2.2 Sistemi numerici

Il **sistema numerico binario** è equivalente al **sistema numerico decimale** che utilizziamo usualmente per rappresentare i numeri interi. Il sistema numerico decimale utilizza dieci cifre (`0, 1, 2, 3, 4, 5, 6, 7, 8` e `9`) mentre il sistema numerico binario solo due cifre (`0` e `1`) ma è comunque capace di rappresentare qualsiasi numero intero.

Il sistema numerico decimale e il sistema numerico binario vengono anche detti **sistema numerico posizionale in base 10**, e **sistema numerico posizionale in base 2.** Infatti, in entrambi i sistemi ogni cifra di un numero viene interpretata in base alla sua posizione. Partendo da destra verso sinistra e considerando la posizione iniziale con indice `0`, avremo che:

1. per interpretare un numero nel sistema decimale, ogni cifra nella posizione `n` deve essere moltiplicato per 10^n, e i risultati delle moltiplicazioni vanno poi sommati tra loro. Per esempio, consideriamo il numero `127`. Tenendo presente che `1` è alla posizione `2`, `2` è alla posizione `1`, e `7` è alla posizione `0`, risulterà che `127` è uguale a $1*10^2 + 2*10^1 + 7*10^0$ che è uguale a `100 + 20 + 7`.
2. Per interpretare un numero nel sistema binario, ogni cifra nella posizione `n` deve essere moltiplicato per 2^n, e i risultati delle moltiplicazioni vanno poi sommati tra loro. Per rappresentare `127` con il sistema binario quindi, dovremo scrivere `1111111`. Infatti $1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$ è uguale a `64 + 32 + 16 + 8 + 4 + 2 + 1 = 127`.

Esistono anche altri sistemi come il **sistema numerico ottale** ed il **sistema numerico esadecimale**, detti anche rispettivamente **sistema numerico posizionale in base 8**, e **sistema numerico posizionale in base 16.** Il sistema numerico ottale utilizza solo otto cifre (`0, 1, 2, 3, 4, 5, 6, 7`), mentre

l'esadecimale ben sedici (`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F`). Per questi sistemi valgono le stesse considerazioni fatte per il sistema decimale e il sistema binario. Quindi col sistema ottale `127` sarà scritto `177` (infatti $1*8^2 + 7*8^1 + 7*8^0 = 64 + 56 + 7 = 127$). Invece in esadecimale scriveremo `7F` (infatti $7*16^1 + F*16^0 = 112 + 15 = 127$).

3.2.3 Complemento a due

Java per immagazzinare i numeri con segno all'interno dei byte, utilizza il metodo chiamato **complemento a due** (in inglese **two's complement).** Questo metodo stabilisce le regole che il processore deve usare per la rappresentazione dei numeri con segno. Con il complemento a due in un singolo byte (ovvero 8 bit) riusciremo a rappresentare un numero intero tra 256 possibilità. In particolare questo numero intero sarà compreso tra 128 numeri positivi e 128 negativi. Quindi l'**intervallo di rappresentazione** di un byte è compreso tra i numeri `-128` a `+127` (lo `0` è considerato un numero positivo). Infatti, il primo bit (quello più a sinistra), viene detto **bit del segno** (o anche **bit più significativo**) perché viene utilizzato per dichiarare il segno del numero definito dai bit seguenti. Se questo bit contiene il valore `0` vuol dire che il numero rappresentato dai successivi bit è positivo, mentre se vale `1`, vuole dire che è negativo. Quindi se il bit del segno è `0` equivale al simbolo `+`, mentre se è `1` equivale al simbolo `-`. Utilizzato un bit per il segno, in un byte ci restano 7 bit per poter rappresentare un numero. Come abbiamo visto nell'esempio del paragrafo precedente, il numero più alto rappresentabile con 7 bit è `127`, rappresentato come `1111111`. In definitiva il valore più alto che si può rappresentare in un byte è `01111111`, dove `0` è il bit del segno e vale `+`.

Per rappresentare un numero negativo con il complemento a due, bisogna invertire tutti i valori dei bit del relativo numero positivo e poi sommare il valore `1`. Per esempio consideriamo il numero `9` che in binario ha la seguente rappresentazione:

$$00001001\ ([+]\ 0*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = +\ 0+0+0+8+0+0+1 = +9)$$

Per ottenere il valore `-9`, bisogna prima invertire tutti i valori dei bit

ottenendo quello che viene detto “complemento a uno” (che è un altro metodo di rappresentazione dei numeri con segno) `11110110`. Poi bisogna sommare `1` (che in binario si scrive `00000001`) a questo risultato ottenendo `11110111`, che è la rappresentazione di `-9` in complemento a due. Notare che il bit del segno questa volta vale `1`, il che implica il segno `-` davanti al `9`.

Se vogliamo immagazzinare un numero superiore a `127`, o minore di `-128`, automaticamente il dato si estenderà su di un’area di memoria più estesa che comprenderà più byte, ma che considererà come bit del segno, il primo bit del primo byte (quello più a sinistra). Per esempio se volessimo immagazzinare il valore `2004`, allora ci occorrerebbero due byte (16 bit): `0000011111010100`, dove il primo bit è il bit del segno (positivo).

Il complemento a due è il metodo di rappresentazione più utilizzato nell’informatica moderna per la sua efficienza in alcuni tipi di operazioni, ma non approfondiremo l’argomento oltre, non essendo questa la sede adatta per farlo. Infatti con Java non avremo mai a che fare con la memoria direttamente. Questi ultimi argomenti sono stati trattati solo per comprendere meglio come vengono rappresentati i tipi di dati in Java.

3.3 Tipi di dati primitivi

Java definisce solamente otto **tipi di dati primitivi**, ovvero tipi di dati semplici che vedremo si differenziano dai tipi complessi. Come abbiamo già visto, vengono utilizzati per:

- tipi interi: `byte, short, int, long`.
- Tipi floating point (o a virgola mobile): `float` e `double`.
- Tipo testuale: `char`.
- Tipo logico-booleano: `boolean`.

> **In altri linguaggi di programmazione esistono tipi senza segno (unsigned), che invece Java non possiede.**

3.3.1 Tipi di dati interi

I **tipi di dati interi** sono quattro: `byte`, `short`, `int` e `long`. Essi condividono la stessa funzionalità (tutti possono immagazzinare numeri interi positivi o negativi) ma differiscono per quanto riguarda il proprio intervallo di rappresentazione. Infatti un `byte` può immagazzinare un intero utilizzando un byte (otto bit), uno `short` due byte (16 bit), un `int` quattro byte (32 bit) ed un `long` otto byte (64 bit). Lo schema seguente riassume dettagliatamente i vari intervalli di rappresentazione:

Tipo	Intervallo di rappresentazione
`byte`	8 bit (da -128 a +127)
`short`	16 bit (da -32.768 a +32.767)
`int`	32 bit (da -2.147.483.648 a +2.147.483.647)
`long`	64 bit (da -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807)

Si noti che l'intervallo di rappresentazione dei tipi primitivi interi è sempre compreso tra un minimo di -2 elevato al numero di bit meno uno, ed un massimo di 2 elevato al numero di bit meno uno, sottraendo alla fine un'unità. Quindi potremmo riscrivere la tabella precedente nel seguente modo:

Tipo	Intervallo di rappresentazione
`byte`	8 bit (da -2^7 a 2^7-1)
`short`	16 bit (da -2^{15} a $2^{15}-1$)
`int`	32 bit (da -2^{31} a $2^{31}-1$)
`long`	64 bit (da -2^{63} a $2^{63}-1$)

Si noti anche che per ogni tipo numerico esiste un numero pari di numeri positivi e di numeri negativi. Infatti il numero `0` è considerato un numero positivo.

Per immagazzinare un intero in Java, è possibile utilizzare quattro tipi di sistemi numerici: decimale (o notazione naturale), binaria, ottale ed esadecimale.

Per la notazione binaria bisogna anteporre al numero intero uno `0` (zero) e una `b` (maiuscola o minuscola). Per la notazione ottale è necessario anteporre al numero intero uno `0` (zero). Per la notazione esadecimale `0` e `x` (indifferentemente maiuscola o minuscola). Per la notazione decimale (o naturale) non c'è bisogno di utilizzare prefissi. Segue qualche esempio d'utilizzo di tipi interi:

```
byte b = 10; //notazione decimale: b vale 10
short s = 022; //notazione ottale: s vale 18
long l = 0x12acd; //notazione esadecimale: l vale 76493
int i = 1000000000; //notazione decimale: i vale 1000000000
int n = 0b10100001010001011010000101000101 //notazione
binaria:
                n vale -1589272251
```

Ci sono alcune osservazioni da fare riguardo i tipi di dati primitivi. Facciamo subito un esempio considerando la seguente assegnazione:

```
byte b = 127;
```

Il compilatore è in grado di capire che `127` è un numero appartenente all'intervallo di rappresentazione dei `byte` e quindi l'espressione precedente è corretta e compilabile senza problemi. Al contrario le seguenti istruzioni:

```
byte b = 128; //il massimo per byte è 127
short s = 32768; //il massimo per short è 32767
int i = 2147483648; //il massimo per int è 2147483647
```

provocano errori in compilazione.

3.3.1.1 Promozione automatica nelle espressioni

Consideriamo ora la seguente istruzione:

```
byte b = 50;
```

Questo statement è corretto. Il numero intero `50` è tranquillamente compreso nell'intervallo di rappresentazione di un `byte`, che va da `-128` a `+127`. Il compilatore determina la grandezza del valore numerico e controlla se è compatibile con il tipo di dato dichiarato.

Adesso consideriamo quest'altro statement:

```
b = b * 2;
```

Ciò darà luogo ad un errore in compilazione! Infatti il compilatore non eseguirà l'operazione di moltiplicazione per controllare la compatibilità con il tipo di dato dichiarato. Invece promuoverà automaticamente i due operandi ad `int`. Quindi, se `50*2` è un `int`, non può essere immagazzinato in `b` che è un `byte`. Il fenomeno appena descritto è conosciuto come **promozione automatica nelle espressioni** (in inglese sinteticamente **promotion).** Per gli operatori binari esistono quattro regole, che dipendono dai tipi degli operandi in questione:

1. se uno degli operandi è `double`, l'altro operando sarà convertito in `double;`
2. se il più "ampio" degli operandi è un `float`, l'altro operando sarà convertito in `float;`
3. se il più "ampio" degli operandi è un `long`, l'altro operando sarà convertito in `long;`
4. in ogni altro caso entrambi gli operandi saranno convertiti in `int`.

La promozione automatica ad intero degli operandi avviene prima che venga eseguita una qualsiasi operazione binaria.

3.3.1.2 Casting

Ma, evidentemente, `50*2` è immagazzinabile in un `byte`. Esiste una tecnica

per forzare una certa quantità ad essere immagazzinata in un certo tipo di dato. Questa tecnica è nota come **cast** (o **casting**). La sintassi da utilizzare per risolvere il nostro problema è:

```
  b = (byte)(b * 2);
```

ovvero bisogna anteporre all'espressione da convertire (volgarmente "castare") il tipo di dato a cui vogliamo restringere il risultato tra due parentesi tonde. In questo modo il compilatore sarà avvertito che un'eventuale perdita di precisione è calcolata e sotto controllo. Bisogna essere però molto prudenti nell'utilizzare il casting in modo corretto. Infatti se scrivessimo:

```
  b = (byte)128;
```

il compilatore non segnalerebbe nessun tipo d'errore. Siccome il cast agisce troncando i bit in eccedenza. Nel nostro caso, dato che un `int` utilizza 32 bit, mentre un `byte` solo 8, saranno troncati i primi 24 bit dell'`int`. Quindi, dato che la rappresentazione del numero `128` all'interno di un tipo int (32 bit) è la seguente:

```
00000000000000000000000010000000
```

troncando i primi 24 bit avremo:

```
10000000
```

che rappresenta il numero `-128`, quindi la variabile `b` avrà il valore di `–128` e non di `128`!

3.3.1.3 Operazioni con interi

Un altro tipico problema di cui preoccuparsi è la **somma di due interi.** Per le stesse ragioni di cui sopra, se la somma di due interi supera il range consentito, è comunque possibile immagazzinarne il risultato in un intero senza avere errori in compilazione, ma il risultato sarà diverso da quello previsto. Per esempio, le seguenti istruzioni saranno compilate senza errori:

```
int a = 2147483647; // Massimo valore per un int
int b = 1;
int risultato = a+b;
```

ma il valore della variabile risultato sarà di `-2147483648`!

Anche la **divisione tra due interi** rappresenta un punto critico! Infatti il risultato finale, per quanto detto sinora, non potrà che essere immagazzinato in un intero, ignorando così eventuali cifre decimali. Quindi in Java, a meno di cast esplicito, `3` diviso `2` darà come risultato `1` e non `1.5` come ci si aspetterebbe.

Inoltre, se utilizziamo una variabile `long`, a meno di cast espliciti, essa sarà sempre inizializzata con un intero. Quindi, se scriviamo:

```
long l = 2000;
```

dobbiamo tener ben presente che `2000` è un `int` per default, ma il compilatore non ci segnalerà errori perché un `int` può essere tranquillamente immagazzinato in un `long`. Per la precisione dovremmo scrivere:

```
long l = 2000L;
```

che rappresenta con una **sintassi più compatta** il casting da `int` a `long`. Quindi, un cast a `long` si ottiene con una sintassi diversa dal solito, posponendo una "elle" maiuscola o minuscola al valore intero assegnato.

Si preferisce utilizzare la notazione maiuscola, dato che una "elle" minuscola si può confondere con il numero "uno" in alcuni ambienti di sviluppo. Notare che saremmo obbligati ad un casting a `long` nel caso in cui volessimo assegnare alla variabile `l` un valore fuori dell'intervallo di rappresentazione di un `int`. Per esempio:

```
long l = 3000000000;
```

produrrebbe un errore in compilazione. Bisogna eseguire il casting nel seguente modo:

```
long l = 3000000000L;
```

Nonostante tutto, l'`int` è il tipo intero che viene usato più spesso.

3.3.2 Tipi di dati a virgola mobile, casting e promotion

Per memorizzare i numeri decimali Java utilizza i tipi a **virgola mobile** (in inglese **floating point).** Si parla di virgola mobile perché un numero decimale è possibile rappresentarlo spostando la virgola utilizzando la notazione scientifica. Per esempio, `0,0001` è equivalente a `1 * 10`$^{-4}$, che con la notazione scientifica si scrive `1E4`. I due tipi a virgola mobile che Java supporta con i loro intervalli di rappresentazione (basati sullo standard di decodifica IEEE-754) sono riportati di seguito:

Operatore	Intervallo di rappresentazione
`float`	32 bit (da +/-1.40239846E-45 a +/-3.40282347E38)
`double`	64 bit (da +/-4.94065645841246544E-324 a +/-1.79769313486231570E328)

Negli Stati Uniti viene usato come separatore decimale (il simbolo che separa la parte intera da quella frazionaria) il simbolo di punto. Anche in Java si userà quindi tale simbolo. Noi italiani invece siamo abituati ad utilizzare il simbolo di virgola. Quindi per esempio `12,3` in Java si scriverà `12.3`. Questo spiega anche perché abbiamo tradotto "floating point" (letteralmente punto fluttuante) con "virgola mobile".

Un primo esempio di dichiarazione di `double` potrebbe essere la seguente:

```
double d = 0.0126;
```

È anche possibile utilizzare la notazione esponenziale o ingegneristica (la "e" può essere sia maiuscola sia minuscola), per esempio:

```
  double d = 1.26E-2; //equivalente a 1.26 diviso 100 = 0.0126
```

Per quanto riguarda cast e promotion la situazione cambia rispetto al caso dei tipi interi. Mentre i valori interi sono assunti dal compilatore come `int` (tipo a 32 bit) di default, per i valori floating point è considerato default il tipo **double** (64 bit) e non **float** (32 bit) come ci si potrebbe aspettare. Ciò implica che se vogliamo assegnare un valore a virgola mobile ad un **float** non possiamo fare a meno di un cast. Per esempio, la seguente riga di codice provocherebbe un errore in compilazione:

```
  float f = 3.14;
```

Anche in questo caso, il linguaggio ci viene incontro permettendoci il cast con la sintassi breve:

```
  float f = 3.14F;
```

La "effe" può essere sia maiuscola sia minuscola.

Esiste, per quanto ridondante, anche la forma contratta del cast per i `double`:

```
  double d = 10.12E24;
```

che è equivalente a scrivere:

```
  double d = 10.12E24D;
```

Alcune operazioni matematiche potrebbero dare risultati che non sono compresi nell'insieme dei numeri reali (per esempio "infinito"). Nella libreria standard sono definite delle classi dette **classi wrapper** (in italiano **classi involucro)**, che non sono altro che classi che rappresentano i tipi di dati primitivi. Per esempio esiste la classe `Integer` per il tipo primitivo `int`, la classe `Short` per il tipo di dato `short` e così via. Gli oggetti istanziati da queste classi sono interscambiabili con i dati primitivi grazie alla caratteristica di Java nota come **autoboxing-autounboxing.** È per questo che le classi wrapper `Double` e `Float`, forniscono le seguenti costanti statiche:

- `Float.NaN`
- `Float.NEGATIVE_INFINITY`
- `Float.POSITIVE_INFINITY`
- `Double.NaN`
- `Double.NEGATIVE_INFINITY`
- `Double.POSITIVE_INFINITY`

Dove `NaN` sta per “Not a Number” (“non un numero”). Per esempio:

```
double d = -10.0 / 0.0;
System.out.println(d);
```

produrrà il seguente output:

```
NEGATIVE_INFINITY
```

Incontreremo le classi wrapper più volte durante la lettura di questo libro.

> **È bene aprire una piccola parentesi sui modificatori `final` e `static`. Infatti, una costante statica in Java è una variabile dichiarata `final` e `static`. In particolare, il modificatore `final` applicato ad una variabile farà in modo che ogni tentativo di cambiare il valore di tale variabile (già inizializzata) produrrà un errore in compilazione. Quindi una variabile `final` è a tutti gli effetti una costante. Per quanto riguarda il modificatore `static`, come abbiamo già avuto modo di asserire precedentemente, il discorso è più complicato. Quando dichiariamo una variabile statica, tale variabile sarà condivisa da tutte le istanze di quella classe, ovvero è come se appartenesse proprio alla classe invece che ai suoi oggetti.**
>
> **Ecco perché accediamo alle costanti statiche tramite la sintassi `NomeClasse.nomeCostanteStatica` (per esempio `Float .NaN`) invece di `NomeClasse.nomeCostanteStatica`. I**

modificatori `static` **e** `final`, **saranno trattati in dettaglio nel capitolo 5.**

Sembra evidente che non sia così banale eseguire calcoli con i tipi primitivi. Spesso conviene utilizzare solamente tipi `double` in caso di espressioni aritmetiche che coinvolgono numeri decimali. Purtroppo però, persino utilizzando solo variabili `double`, non saremo sicuri di ottenere risultati precisi in caso di espressioni che coinvolgono più cifre decimali. Infatti, avendo i `double` una rappresentazione numerica comunque limitata, a volte devono essere arrotondati. Inoltre l'arrotondamento dipende dall'architettura dalla piattaforma su cui viene eseguito il calcolo.

Esiste in Java un modificatore che viene usato rarissimamente, si tratta di `strictfp`, **e probabilmente la maggior parte dei programmatori Java non ne ha mai fatto uso. Dichiarare le variabili** `float` **e** `double` **come** `strictfp`, **permetterebbe di troncare i bit che farebbero variare questi arrotondamenti su piattaforme diverse, garantendo così uniformità di risultati su tutte le piattaforme (purtroppo i risultati potrebbero comunque essere sbagliati!). Il modificatore** `strictfp` **può marcare non solo le variabili** `float` **e** `double`, **ma anche classi o metodi. In questi casi tutte le operazioni fatte all'interno di queste classi o metodi seguiranno le regole del troncamento di** `strictfp`.

Per ottenere risultati precisi nelle operazioni è necessario utilizzare una classe della libreria Java: `BigDecimal` (package `java.math;` cfr. Documentazione ufficiale) in luogo del tipo `double`.

3.3.3 Underscore in tipi di dati numerici

Per migliorare la leggibilità dei valori assegnati alle nostre variabili, è possibile usare anche i simboli di "_" **(simbolo di sottolineatura** o

underscore). Per esempio:

```
int i = 1000000000;
```

```
int n = 0b10100001010001011010000101000101
```

possono essere riportati come segue:

```
int i = 1_000_000_000;
int n = 0b10100001_01000101_10100001_01000101
```

Quando si vogliono utilizzare gli underscore all'interno dei numeri bisogna sapere che non è possibile usarli:

1. all'inizio o alla fine di un numero;
2. adiacenti ad un punto decimale per i tipi di dati a virgola mobile;
3. prima dei suffissi `F` o `L` (sia maiuscole che minuscole) che vengono usati per i cast a `float` e `long`;
4. nelle posizioni dove ci si aspetta una stringa di caratteri.

Con qualche esempio preso direttamente dalla documentazione del JDK faremo un po' di chiarezza:

```
float piGreco = 3.14_15F;
long bytesEsadecimali = 0xFF_EC_DE_5E;
long maxLong = 0x7fff_ffff_ffff_ffffL;
long bytes = 0b11010010_01101001_10010100_10010010;
float pi1 = 3_.1415F;     // Errore: violata regola 2
float pi2 = 3._1415F;     // Errore: violata regola 2
long socialSecurityNumber = 999_99_9999_L; // Errore:
violata regola 3
int x1 = _52;              // Errore: violata regola 1
int x2 = _5_2;
int x3 = 52_;             // Errore: violata regola 1
int x4 = 5_____2;
int x5 = 0_x52;           // Errore: violata regola 4
int x6 = 0x_52;           // Errore: violata regola 1
int x7 = 0x5_2;
int x8 = 0x52_;           // Errore: violata regola 1
int x9 = 0_52;            // OK! Non viola la regola 1
    (notare che si tratta di rappresentazione ottale)
```

3.3.4 Tipo di dato logico - booleano

Il tipo di dato `boolean` utilizza solo un bit per memorizzare un valore e gli unici valori che può immagazzinare sono `true` e `false`. Per esempio:

```
boolean b = true;
```

Vengono definiti **literals** i valori contenenti lettere o simboli che vengono specificati nel codice sorgente invece che al runtime. Possono rappresentare variabili primitive o stringhe e possono apparire solo sul lato destro di un'assegnazione o come argomenti quando si invocano metodi. Non è possibile assegnare valori ai literals. Chiaramente i literals devono contenere caratteri o simboli (come per esempio il punto). Esempi di literals sono `true` e `false` (gli unici due valori assegnabili ad un tipo booleano) ma anche i valori assegnati alle stringhe, ai caratteri, e ai numeri rappresentati con simboli e lettere. Segue qualche esempio di literals:

```
boolean isBig = true;
char c = 'w';
String a = "\n";
int i = 0x1c;
float f = 3.14f;
double d = 3.14;
int unMiliardo = 1_000_000_000;
```

La definizione di literal potrebbe risultare fine a sé stessa, ma è una di quelle definizioni che spesso viene menzionata nella documentazione standard.

3.3.5 Tipo di dato primitivo letterale

Il tipo `char` permette di immagazzinare caratteri (uno per volta). Questo tipo di dato viene memorizzato in 2 byte (16 bit) e utilizza lo standard di decodifica dei caratteri noto come **Unicode.** Java 9 supporta la versione 8.0 di Unicode.

Possiamo affermare che Unicode può essere utilizzato in tre modi:

1. UTF-8: a 8-bit che coincide con la codifica ASCII.
2. UTF-16: a 16 bit, che contiene tutti i caratteri della maggior parte degli alfabeti del mondo. Anche il `char` è 16 bit e quindi con esso siamo in grado di rappresentare praticamente tutti i caratteri più significativi esistenti.
3. UTF-32: a 32 bit che contiene codifiche di altri caratteri compresi che potrebbero essere considerati meno usati. Java supporta l'utilizzo di UTF-32 con uno stratagemma basato su un `int` (ovvero bisogna concatenare due caratteri Unicode), ma sarà molto raro avere bisogno di utilizzare questa forma.

Per maggiori informazioni, **http://www.unicode.org**.

Possiamo assegnare ad un `char` un qualsiasi carattere che si trova sulla nostra tastiera (ma anche il sistema operativo deve supportare il carattere richiesto). Oppure possiamo assegnare ad un `char` direttamente un valore Unicode in formato esadecimale, che identifica univocamente un determinato carattere, anteponendo ad esso il prefisso `\u`. In qualsiasi caso, dobbiamo comprendere tra apici singoli il valore da assegnare. Ecco qualche esempio:

```
char primoCarattere = 'a';
char car = '@';
char letteraGrecaPhi = '\u03A6'; //(lettera "Φ")
char carattereUnicodeNonIdentificato = '\uABC8';
```

Esiste anche la possibilità di immagazzinare **caratteri di escape**, ovvero sequenze di caratteri che provocano particolari comportamenti nella stampa:

- `\n` che equivale ad andare a capo (tasto new line)
- `\\` che equivale ad un solo `\` (tasto backslash)
- `\t` che equivale ad una tabulazione (tasto TAB)
- `\'` che equivale ad un apice singolo
- `\"` che equivale ad un doppio apice (virgolette)

Si noti che è possibile utilizzare caratteri in espressioni aritmetiche. Per esempio, possiamo sommare un `char` con un `int`. Infatti, ogni carattere viene decodificato tramite Unicode a partire da un numero intero. Per esempio:

```
int i = 1;
char a = 'A';
char b = (char)(a+i); // b ha valore
'B'!
```

3.4 Tipi di dati non primitivi: reference

Abbiamo già visto come istanziare oggetti da una certa classe. Dobbiamo prima dichiarare un oggetto di tale classe con una sintassi di questo tipo:

```
NomeClasse nomeOggetto;
```

per poi istanziarlo utilizzando la parola chiave `new`.

Dichiarare un oggetto quindi è del tutto simile a dichiarare un tipo di dato primitivo. Il "nome" che diamo ad un oggetto è detto **reference.** Infatti non si sta parlando di una variabile tradizionale bensì di una variabile che alcuni definiscono **puntatore.** Possiamo definire un puntatore come una variabile che contiene un indirizzo in memoria.

C'è una sottile e potente differenza tra la dichiarazione di un tipo di dato primitivo ed uno non primitivo. Consideriamo un esempio partendo dalla definizione di una classe che astrae in maniera banale il concetto di data:

```
public class Data {
    public int giorno;
    public int mese;
    public int anno;
}
```

Per il nostro esempio `Data` sarà quindi un tipo di dato non primitivo (astratto). Come tipo di dato primitivo prendiamo ad esempio un `double`, e consideriamo le seguenti righe di codice supponendo che si trovino

all’interno di un metodo `main()` di una classe qualsiasi:

```
double unNumero = 5.0;
Data unGiorno = new Data();
```

Potrebbe aiutarci una schematizzazione di come saranno rappresentati i dati in memoria quando l’applicazione è in esecuzione, ma riportare fedelmente queste informazioni è un’impresa ardua e forse anche inutile. Infatti non avremo mai a che fare direttamente con la gestione della memoria se programmiamo in Java.

Ad ogni modo può essere utile immaginare la situazione in memoria con il tipo di schematizzazione, frutto di una convenzione, illustrato in figura 3.2.

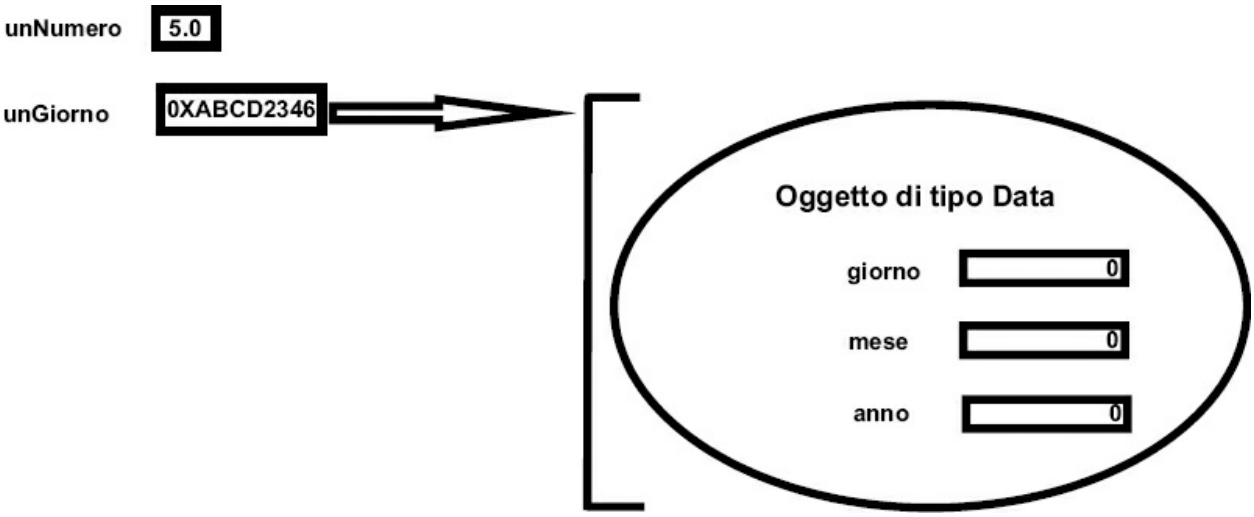


Figura 3.2 - Convenzione di schematizzazione della memoria.

La differenza pratica tra un reference ed una variabile, diviene evidente nelle assegnazioni. Consideriamo il seguente **snippet**, ovvero un frammento di codice non contestualizzato:

```
double unNumero = 5.0;
double unAltroNumero = unNumero;
Data unGiorno = new Data();
Data unAltroGiorno = unGiorno;
```

Sia per il tipo di dato primitivo sia per quello complesso abbiamo creato un comportamento equivalente: dichiarazione ed assegnazione di un valore, e riassegnazione di un altro valore.

La variabile `unAltroNumero` assumerà lo stesso valore della variabile `unNumero`, ma le due variabili rimarranno indipendenti l'una dall'altra. Il valore della variabile `unNumero` verrà infatti copiato nella variabile `unAltroNumero`. Se il valore di una delle due variabili sarà modificato in seguito, il valore dell'altra variabile non cambierà.

Il reference `unAltroGiorno`, invece, assumerà semplicemente il valore (cioè l'indirizzo) del reference `unGiorno`. Ciò significa che `unAltroGiorno` punterà allo stesso oggetto cui punta `unGiorno`. La figura 3.3 mostra la situazione rappresentata graficamente.

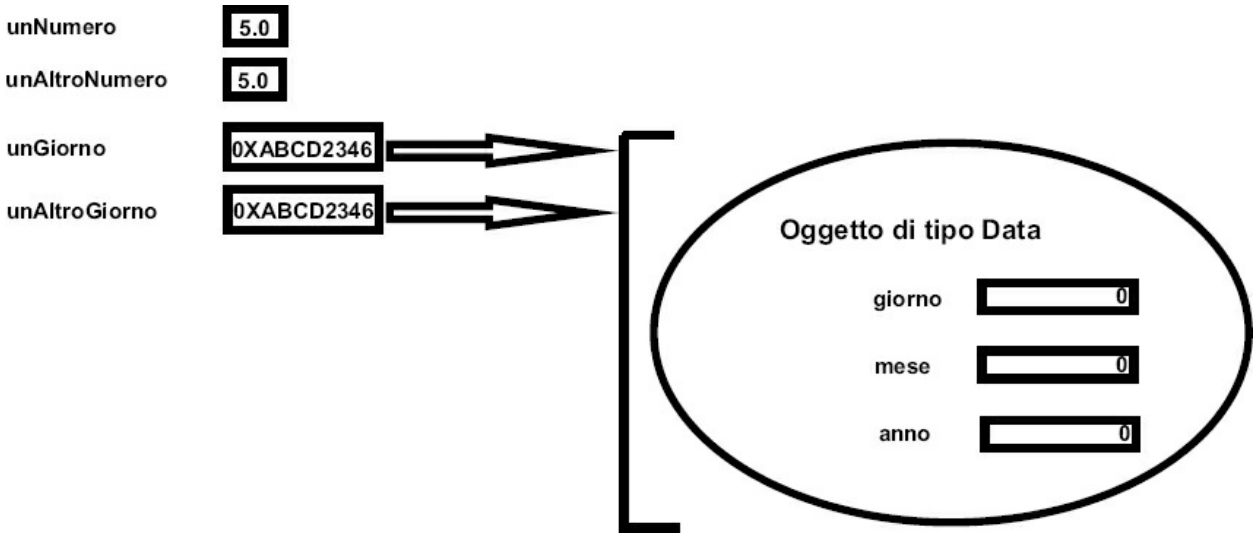


Figura 3.3 - Convenzione di schematizzazione della memoria.

Quindi, se in seguito sarà apportata una qualche modifica all'oggetto comune tramite uno dei due reference, questa sarà verificabile anche tramite l'altro reference. Per intenderci:

```
unGiorno.anno
```

è sicuramente equivalente a:

```
unAltroGiorno.anno
```

3.4.1 Passaggio di parametri per valore

“Il passaggio di parametri in Java avviene sempre per valore”.

Quest’affermazione viene contraddetta in alcuni testi ma basterà leggere l’intero paragrafo per fugare ogni dubbio.

Quando si invoca un metodo che come parametro prende in input una variabile, al metodo stesso viene passato solo il valore (una copia) della variabile che quindi rimane immutata anche dopo l’esecuzione del metodo. Per esempio consideriamo la classe:

```
public class CiProvo {
    public void cambiaValore(int valore) {
        valore = 1000;
    }
}
```

il seguente frammento di codice (snippet):

```
CiProvo ogg = new CiProvo();
int numero = 10;
ogg.cambiaValore(numero);
System.out.println("il valore del numero è " + numero);
```

produrrà il seguente output:

```
il valore del numero è 10
```

Infatti il parametro `valore` del metodo `cambiaValore()`, nel momento in cui è stato eseguito il metodo, non coincideva con la variabile `numero`, bensì immagazzinava solo la copia del suo valore (10). Quindi ovviamente la variabile `numero` non è stata modificata.

Stesso discorso vale per i tipi reference: viene sempre passato il valore del reference, ma come abbiamo detto il valore di un reference coincide con l’indirizzo in memoria. Consideriamo la seguente classe:

```
public class CiProvoConIReference {
```

```
    public void cambiaReference(Data data) {
        data = new Data();
        // Un oggetto appena istanziato
        // ha le variabili inizializzate a valori nulli
    }
}
```

Il seguente frammento di codice:

```
CiProvoConIReference ogg = new CiProvoConIReference();
Data dataDiNascita = new Data();
dataDiNascita.giorno = 14;
dataDiNascita.mese = 4;
dataDiNascita.anno = 2004;
ogg.cambiaReference(dataDiNascita);
System.out.println("Data di nascita = "
  + dataDiNascita.giorno + "-" + dataDiNascita.mese
  + "-" +dataDiNascita.anno);
```

produrrà il seguente output:

```
Data di nascita = 14-4-2004
```

Valgono quindi le stesse regole anche per i reference.

Se il metodo `cambiaReference()` avesse cambiato i valori delle variabili d'istanza dell'oggetto avremmo avuto un output differente. Riscriviamo il metodo in questione:

```
public void cambiaReference(Data data) {
    data.giorno = 12; // data punta allo stesso indirizzo
    data.mese = 11; // della variabile dataDiNascita
    data.anno = 2006;
}
```

Il fatto che il passaggio avvenga sempre per valore garantisce che un oggetto possa essere modificato all'interno del metodo chiamato, e contemporaneamente si è certi che dopo la chiamata del metodo il reference punti sempre allo stesso oggetto. In altri linguaggi, come il C, è permesso anche il passaggio di parametri "per riferimento". In quel caso al metodo viene passato l'intero riferimento, non solo il suo indirizzo, con la conseguente possibilità di poterne cambiare l'indirizzamento. Questa caratteristica non è stata importata in Java, perché considerata (a ragione) una

minaccia alla sicurezza. Molti virus, worm e simili, sfruttano infatti la tecnica del passaggio per riferimento.

In questo caso Java ha scelto la strada della robustezza e della semplicità, favorendola rispetto alla potenza del linguaggio.

> **Alcuni autori di altri testi affermano che il passaggio di parametri in Java avviene per valore per i tipi di dato primitivi, e per riferimento per i tipi di dato complesso. Chi vi scrive ribadisce che si tratta solo di un problema di terminologia. Probabilmente, se ignorassimo il linguaggio C, anche noi daremmo un significato diverso al "passaggio per riferimento". L'importante è capire il concetto senza fare confusione.**

3.4.2 Inizializzazione delle variabili d'istanza

Abbiamo già asserito che nel momento in cui è istanziato un oggetto, tutte le sue variabili d'istanza (che con esso condividono il ciclo di vita) vengono inizializzate ai rispettivi valori nulli. Di seguito è presentata una tabella che associa ad ogni tipo di dato il valore con cui viene inizializzata una variabile di istanza al momento della sua creazione:

Variabile	Valore
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
`char`	`'\u0000'` (**Valore nullo - nessun carattere**)
`boolean`	`false`

Ogni tipo reference	`null`

È abbastanza logico considerare come valori nulli quelli presenti in questa tabella. Si possono notare le notazioni di cast vicino ai valori nulli di `long`, `float` e `double`. Inoltre il valore nullo del `char` è espresso in notazione esadecimale e rappresenta "nessun carattere". Infine per ogni tipo reference il valore letterale `null` implica che il reference non ha un indirizzo a cui puntare, e punta all'indirizzo `null` (dove non ci sono oggetti). Purtroppo vedremo che un tipico errore nella programmazione Java (probabilmente il più comune) riguarda proprio la nullità dei reference.

3.5 Introduzione alla libreria standard

Come già accennato più volte, Java possiede un'enorme e lussuosa libreria di classi standard che costituisce uno dei punti di forza del linguaggio. È anche nota come **Java API**, dove API è l'acronimo per "Application Programming Interface" (letteralmente "Interfaccia di programmazione di un'applicazione"). Essa è organizzata in vari package che raccolgono le classi secondo un'organizzazione basata sul campo d'utilizzo. I principali package "storici" (ovvero che esistono dalla prima versione di Java) sono i seguenti:

- `java.io` contiene classi per realizzare l'input - output in Java.
- `java.awt` contiene classi per realizzare interfacce grafiche, come `Button` (caduto quasi completamente in disuso).
- `java.net` contiene classi per realizzare connessioni, come `Socket`.
- `java.applet` contiene un'unica classe: `Applet`. Questa permette di realizzare applet. Con la versione 9 però, questa libreria è stata **deprecata**, ovvero Oracle ha deciso che non rientra più nei piani per il futuro del linguaggio, e che quindi sarà prima o poi eliminata.
- `java.util` raccoglie classi d'utilità, come `Date`.

- `java.lang` è il package che contiene le classi nucleo del linguaggio, come `System` e `String`.
- Affronteremo questi package nei capitoli successivi e in alcune appendici on line.

3.5.1 Il comando `import`

Per utilizzare una classe della libreria all'interno di una classe che abbiamo intenzione di scrivere, bisogna prima importarla. Supponiamo di voler utilizzare la classe `Date` del package `java.util`. Prima di dichiarare la classe in cui abbiamo intenzione di utilizzare `Date`, dobbiamo scrivere:

```
import java.util.Date;
```

oppure, per importare tutte le classi del package `java.util`:

```
import java.util.*;
```

Di default in ogni file Java è importato automaticamente tutto il package `java.lang`, senza il quale non potremmo utilizzare classi fondamentali quali `System` e `String`. Notiamo che questo è un altro esempio di come Java sia pensato anche per favorire l'apprendimento iniziale. Quindi, nel momento in cui compiliamo una classe Java, il compilatore anteporrà il comando:

```
import java.lang.*;
```

alla dichiarazione della nostra classe.

> **L'asterisco non implica l'importazione delle classi appartenenti ai "sottopackage"; per esempio** `import java.*` **non include** `java.awt.*` **né** `java.awt.event.*`. **Quindi l'istruzione** `import java.*` **non importa tutti i package fondamentali.**

Per dare un'idea della potenza e della semplicità di Java viene presentata di seguito una semplicissima classe. Istanziando qualche oggetto da alcune

classi del package `java.awt` (una libreria grafica a cui è dedicata solo un'appendice on line perché oramai caduta in disuso) ed assemblandoli con un certo criterio oterremo, con poche righe, una finestra con un bottone. La finestra, sfruttando la libreria `java.awt`, erediterà lo stile grafico del sistema operativo su cui esegue. Quindi sarà visualizzato lo stile di Windows su Windows, lo stile Motif su sistema operativo Solaris e così via. Il lettore può farsi un'idea di come è possibile utilizzare la libreria standard e della sua potenza analizzando il seguente codice:

```
import java.awt.*;

public class FinestraConBottone {
    public static void main(String args[]) {
        Frame finestra = new Frame("Titolo");
        Button bottone = new Button("Cliccami");
        finestra.add(bottone);
        finestra.setSize(200,100);
        finestra.setVisible(true);
    }
}
```

Basta conoscere un po' d'inglese per interpretare il significato di queste righe.

La classe `FinestraConBottone` è stata riportata a puro scopo didattico. La pressione del bottone non provocherà nessuna azione, come nemmeno il tentativo di chiudere la finestra. Solo il ridimensionamento della finestra è permesso perché rientra nelle caratteristiche della classe `Frame`. Quindi, . . . per chiudere l'applicazione bisogna spostarsi sulla prompt DOS da dove la si è eseguita e terminare il processo in esecuzione mediante il comando CTRL-C (premere contemporaneamente i tasti CTRL e "c"). Se utilizzate EJE, premere il pulsante "interrompi processo". Un'altra soluzione è bloccare il processo Java direttamente dal task manager di Windows. Anche se l'argomento vi ha incuriosito, il consiglio è di non perdere tempo nel creare interfacce grafiche inutili ed inutilizzabili: bisogna prima

imparare Java! Alle interfacce grafiche e alla libreria AWT è comunque dedicata un'intera appendice che trovate on line. Inoltre, al nuovo standard per creare interfacce grafiche chiamato JavaFX, è dedicata l'appendice S.

3.5.2 La classe `String`

Abbiamo visto che il tipo primitivo `char` permette di immagazzinare un carattere alla volta. In un tipo stringa è invece possibile immagazzinare un testo composto da caratteri, numeri e simboli. Nei nostri programmi utilizzeremo quasi sempre le stringhe e molto più raramente il tipo `char`.

In Java le stringhe sono oggetti, e in quanto oggetti dovrebbero essere istanziate con la solita sintassi tramite la parola chiave `new`. Per esempio:

```
String nome = new String("Mario Rossi");
```

Ovvero sfruttando il concetto di costruttore introdotto nel precedente capitolo. Java però, come spesso accade, semplifica la vita del programmatore permettendogli di utilizzare le stringhe come se si trattasse di un tipo di dato primitivo. Per esempio, possiamo istanziare una stringa nel seguente modo:

```
String nome = "Mario Rossi";
```

che è equivalente a scrivere:

```
String nome = new String("Mario Rossi");
```

Per assegnare un valore ad una stringa bisogna che esso sia compreso tra virgolette, a differenza dei caratteri per cui vengono utilizzati gli apici singoli.

Anche in questo caso possiamo apprezzare le semplificazioni che ci offre Java. Il fatto che sia consentito utilizzare una classe così importante come `String`, come se fosse un tipo di dato primitivo, ci ha permesso d'approcciare ai primi esempi di codice senza l'ulteriore difficoltà di capire come funziona

un costruttore.

Il fatto che `String` sia una classe ci garantisce una serie di metodi di utilità, semplici da utilizzare e sempre disponibili, per compiere operazioni con le stringhe. Qualche esempio è rappresentato dai metodi `toUpperCase()`, che restituisce la stringa su cui è chiamato il metodo con ogni carattere maiuscolo (per contro esiste anche il metodo `toLowerCase()`), `trim()`, che restituisce la stringa su cui è chiamato il metodo ma senza gli spazi che precedono la prima lettera e quelli che seguono l'ultima, e il fondamentale `equals(String)` che permette di confrontare due stringhe.

La classe `String` è evidentemente una classe particolare. Un'altra caratteristica da sottolineare è che un oggetto `String` è **immutabile.** Questo significa che una volta istanziata una stringa il suo contenuto non può cambiare. I metodi di cui sopra, infatti, non vanno a modificare l'oggetto su cui sono chiamati, semmai ne restituiscono un altro. Per esempio le seguenti righe di codice:

```
String a = "claudio";
String b = a.toUpperCase();
System.out.println(a); // a rimane immutato
System.out.println(b); // b è la stringa maiuscola
```

produrrebbero il seguente output:

```
claudio
CLAUDIO
```

> **In generale se si prova a stampare un reference di una stringa che punta a `null` sarà stampata la stringa `null`.**

3.5.3 La documentazione della libreria standard di Java

Per conoscere la classe `String` e tutte le altre classi, basta andare a consultare la **documentazione.** È disponibile on line (https://docs.oracle.com/javase/9/docs/api/overview-summary.html) ma se

avete seguito l’appendice B per scaricare e installare il JDK dovreste già averla scaricata nella cartella “Docs” all’interno del JDK e aver creato un collegamento sul desktop. Sino alla versione 8 di Java, molti programmatori preferivano non scaricare la documentazione e consultarla direttamente on line. Infatti per trovare la pagina di una certa classe era più semplice eseguire una ricerca su Google che andare ad esplorare l’albero dei package. Dalla versione 9 finalmente anche la documentazione stessa fornisce un campo per effettuare le ricerche nella parte in alto a destra dell’interfaccia come mostrato in figura 3.4.

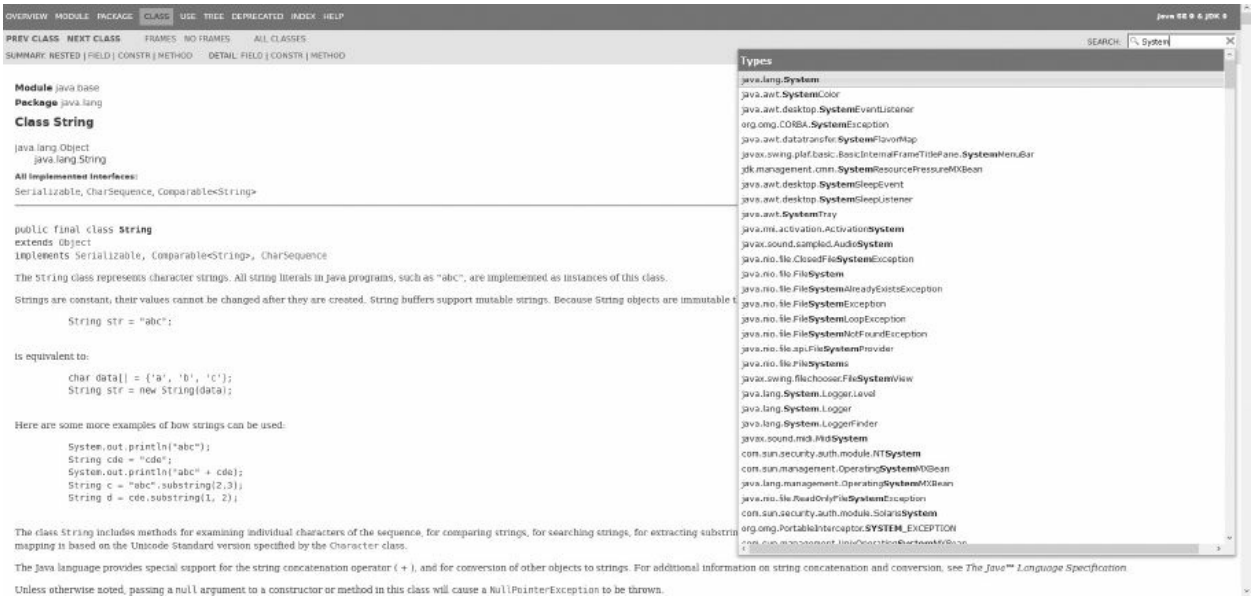


Figura 3.4 - Documentazione di Java 9 con ricerca.

È assolutamente fondamentale infatti, che il lettore inizi da subito la sua esplorazione e conoscenza della documentazione. Il vero programmatore Java ha grande familiarità con essa e sa sfruttarne la facilità di consultazione nel modo migliore. In questo testo, a differenza di altri, saranno affrontati argomenti concernenti le classi della libreria standard, ma in maniera essenziale. Quindi non perderemo tempo nel descrivere tutti i metodi di una classe (tranne che in alcuni casi). Piuttosto ci impegneremo a capire quali sono le filosofie alla base dell’utilizzo dei vari package. Questo essenzialmente perché:

1. riteniamo la documentazione ufficiale insostituibile.

2. Internet è una fonte inesauribile di informazioni ed esempi.
3. Le librerie sono in continua evoluzione.
4. La documentazione è in formato HTML e permette una rapida consultazione. È ovviamente in inglese, e come già detto, nella programmazione un minimo di conoscenza della lingua inglese è richiesto.

> **Se utilizzate EJE, è possibile consultare la documentazione direttamente da EJE. Questo a patto di installare la cartella `docs` all'interno della cartella del JDK (parallelamente a `bin`, `lib`, `jre` etc.). Altrimenti è possibile scegliere la posizione della documentazione in un secondo momento configurandola nelle opzioni.**

3.5.4 Lo strumento javadoc

Nel paragrafo dedicato ai commenti abbiamo accennato alla possibilità di generare la documentazione delle nostre classi in formato HTML, sul modello della documentazione delle classi standard di Java. Infatti esiste un "trucco" per la generazione automatica: lo strumento **javadoc.** Esso permette di generare ipertesti come quello della libreria standard con le informazioni sulle classi che scriveremo. Dobbiamo solamente:

1. scrivere all'interno del commento il codice accompagnato dai commenti da formattare. Bisogna utilizzare commenti del terzo tipo, quelli compresi tra /** e */.
2. Utilizzare il tool javadoc. È molto semplice. Dal prompt, digitare:

```
javadoc nomeFile.java
```

e saranno generati automaticamente tutti i file HTML che servono.

Potremo generare documentazione solo per le classi dichiarate `public`. Possiamo commentare classi, metodi, costruttori, variabili, costanti ed interfacce (se dichiarate `public`). Inoltre il commento deve precedere quello

che si vuol commentare. Per esempio:

```
/**
 Questo è un metodo!
*/
public void metodo() {
    ...
 }
```

> **Con EJE è possibile generare la documentazione javadoc semplicemente facendo clic sul bottone apposito. La documentazione verrà generata in una cartella `docs` creata al volo nella cartella dove si trova il file sorgente.**

3.6 Gli array in Java

Un **array** è una collezione indicizzata di dati primitivi, o di reference o di altri array. Gli array permettono di utilizzare un solo nome per individuare una collezione costituita da vari elementi, che saranno accessibili tramite indici interi. In Java gli array sono, in quanto collezioni, oggetti. La sintassi degli array però, differisce da quella degli altri oggetti.

Per utilizzare un array bisogna passare attraverso tre fasi: dichiarazione, creazione ed inizializzazione.

3.6.1 Dichiarazione

Di seguito presentiamo due dichiarazioni di array. Nella prima dichiariamo un array di `char` (tipo primitivo), nella seconda dichiariamo un array di istanze di `File` (classe appartenente al package `java.io`):

```
char alfabeto [];          oppure          char []
alfabeto;
File files [];            oppure          File [] files;
```

Quindi per dichiarare un array, basta posporre (oppure anteporre) una coppia di parentesi quadre all'identificatore.

3.6.2 Creazione

Un array è un oggetto speciale in Java e, in quanto tale, va istanziato in modo speciale. La sintassi è la seguente:

```
alfabeto = new char[21];
files = new File[3];
```

Come si può notare, è obbligatorio specificare al momento dell'istanza dell'array la dimensione dell'array stesso. A questo punto però tutti gli elementi dei due array sono inizializzati automaticamente ai relativi valori nulli. Vediamo allora come inizializzare esplicitamente gli elementi dell'array.

3.6.3 Inizializzazione

Per inizializzare un array, bisogna inizializzarne ogni elemento singolarmente:

```
alfabeto [0] = 'a';
alfabeto [1] = 'b';
alfabeto [2] = 'c';
alfabeto [3] = 'd';
. . . . . .
alfabeto [20] = 'z';
files [0] = new File("HelloWorld.java");
files [1] = new File("HelloWorld.class");
files [2] = new File("HelloWorld.html");
```

L'indice di un array inizia sempre da zero. Quindi un array dichiarato di 21 posti, avrà come indice minimo `0` e massimo `20` (un array di dimensione `n` implica il massimo indice a `n-1`)

Il lettore avrà sicuramente notato che può risultare alquanto scomodo inizializzare un array in questo modo, per di più dopo averlo prima dichiarato ed istanziato. Ma Java ci viene incontro dando la possibilità di eseguire tutti e tre i passi principali per creare un array tramite una particolare **sintassi breve** che di seguito presentiamo:

```
char alfabeto [] = {'a', 'b', 'c', 'd', 'e', 'z'};
File files [] = {new File("HelloWorld.java"),
```

```
                    new File("HelloWorld.class"),
                    new File("HelloWorld.html")};
```

Si noti la differenza tra un array dichiarato di tipo di dato primitivo o complesso. Il primo contiene direttamente i suoi elementi. Il secondo contiene solo reference, non gli elementi stessi.

Gli array definiscono anche una variabile chiamata `length` che restituisce la dimensione effettiva dell'array stesso. Quindi:

```
alfabeto.length
```

varrà `21`.

Solitamente uno dei vantaggi che porta l'uso di array è sfruttare l'indice all'interno di cicli, che però verranno trattati nel prossimo capitolo.

> **In generale gli array sono utilizzati relativamente poco in Java. Ad essi sono preferiti gli oggetti della libreria Collections che saranno trattati approfonditamente nei capitoli successivi.**

3.6.4 Array Multidimensionali

Esistono anche array multidimensionali che sono array di array. A differenza della maggior parte degli altri linguaggi di programmazione, in Java un array bidimensionale non deve per forza essere rettangolare. Di seguito è presentato un esempio:

```
int arrayNonRettangolare [][] = new int[4][];
arrayNonRettangolare [0] = new int[2];
arrayNonRettangolare [1] = new int[4];
arrayNonRettangolare [2] = new int[6];
arrayNonRettangolare [3] = new int[8];
arrayNonRettangolare [0][0] = 1;
arrayNonRettangolare [0][1] = 2;
arrayNonRettangolare [1][0] = 1;
. . . . . .
arrayNonRettangolare [3][7] = 10;
```

oppure, equivalentemente:

```
int arrayNonRettangolare [][] = {
    {1,2},
    {1,0,0,0},
    {0,0,0,0,0,0},
    {0,0,0,0,0,0,0,10}
};
```

Notare che nella prima riga quando abbiamo istanziato l'oggetto `arrayNonRettangolare`, abbiamo specificato solo la prima dimensione (`4`). Questo significa che ci saranno quattro elementi di questo array, ed ogni elemento sarà a sua volta un altro array di lunghezza ancora da scegliere. Attenzione che una rappresentazione grafica di un array è solo una convenzione che ci aiuta a schematizzare il concetto. Non si dovrebbe quindi parlare di righe e colonne per un array bidimensionale, le righe e le colonne sono solo nella nostra mente. In realtà Java è capace anche di creare array a dieci dimensioni, che noi non potremmo mai rappresentare graficamente.

3.6.5 String `args[]`

Di solito, quando si inizia ad imparare un nuovo linguaggio di programmazione, uno dei primi argomenti che l'aspirante sviluppatore impara a gestire è l'input/output nelle applicazioni. Quando invece s'approccia a Java, rimane misterioso per un certo periodo il comando di output:

```
System.out.println("Stringa da stampare");
```

e resta sconosciuta per un lungo periodo anche un'istruzione che permetta di acquisire dati in input! Ciò è dovuto ad una ragione ben precisa: le classi che permettono di realizzare operazioni di input/output fanno parte del package `java.io` della libreria standard, a cui è dedicato più avanti un apposito capitolo. Questo package è stato progettato con una filosofia ben precisa, basata sul pattern **Decorator** (per informazioni sui pattern cfr. appendice D). Il risultato è un'iniziale difficoltà d'approccio all'argomento, compensata però da una semplicità ed efficacia finale. Per esempio, a un aspirante programmatore può risultare difficoltoso comprendere le ragioni per cui, per stampare una stringa a video, i creatori di Java hanno implementato un meccanismo tanto complesso (`System.out.println()`). Per un

programmatore Java invece è molto semplice utilizzare gli stessi metodi per eseguire operazioni di output complesse, come scrivere in un file o mandare messaggi in rete via socket. Rimandiamo il lettore al capitolo relativo all'input-output per i dettagli. Per non anticipare troppo i tempi, presentiamo intanto un procedimento che consente di dotare di un minimo d'interattività le nostre prime applicazioni. Quando codifichiamo il metodo `main()`, il programmatore è obbligato a fornire una firma che definisca come parametro in input un array di stringhe (di solito chiamato `args`). Questo array immagazzinerà stringhe da riga di comando nel modo seguente. Se per eseguire la nostra applicazione, invece di scrivere a riga di comando:

```
java NomeClassDelMain
```

scrivessimo:

```
java NomeClassDelMain Andrea De Sio Cesari
```

all'interno della nostra applicazione avremmo a disposizione la stringa `args[0]` che ha come valore `Andrea`, la stringa `args[1]` che ha come valore `De`, la stringa `args[2]` che ha come valore `Sio`, e la stringa `args[3]` che ha come valore `Cesari`. Potremmo anche scrivere:

```
java NomeClassDelMain Andrea "De Sio Cesari"
```

In questo modo, all'interno dell'applicazione potremmo utilizzare solamente la stringa `args[0]` che ha come valore `Andrea`, e la stringa `args[1]` che ha come valore `De Sio Cesari`.

Con EJE per poter passare parametri da riga di comando bisogna eseguire l'applicazione dal menu "sviluppo - esegui con argomenti" (in inglese "build - execute with args"). In alternativa è possibile usare la scorciatoia costituita dalla pressione dei tasti MAIUSC-F9. Sarà presentata una maschera per inserire gli argomenti (e solo gli argomenti).

A seconda di quanti argomenti passiamo da riga di comando, la JVM

istanzierà l'array di stringhe args di lunghezza pari al numero di argomenti passati e lo inizializzerà come abbiamo detto. Per esempio il seguente codice:

```
public class TestArgs {
    public static void main(String args[]) {
        System.out.println(args[0]);
    }
}
```

stamperà solo il primo eventuale argomento passato da riga di comando. Se però non passeremo nessun argomento da riga di comando, il programma terminerà con la seguente eccezione:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
      at TestArgs.main(TestArgs.java:3)
```

Studieremo le eccezioni approfonditamente nel capitolo 8. Ma già dal prossimo capitolo, dove studieremo cicli e condizioni, impareremo a scrivere programmi che non si interrompano così facilmente.

Riepilogo

Java è a **schema libero** ma l'indentazione del codice è fondamentale per la leggibilità dei nostri programmi. Supporta tre tipologie di **commenti** (a singola **linea, multilinea** e **Javadoc).** È un linguaggio **case sensitive** e definisce **parole chiave, parole riservate** e **parole a utilizzo limitate**, che non possono essere usate come **identificatori.** Questi possono essere composti solo da determinati **caratteri e simboli** e vanno dichiarati rispettando le **convenzioni standard** di Java.

La memoria è una sequenza di unità di memoria dette **byte.** Ogni byte è composto da 8 **bit** ed ogni bit può immagazzinare un unico valore tra `0` e `1`.

I sistemi numerici **decimale, binario, ottale** ed **esadecimale** sono sistemi posizionali e possono rappresentare qualsiasi dato intero. Java usa il metodo di rappresentazione noto come **complemento a due** per la rappresentazione dei numeri con segno. All'interno di un byte il **bit del segno** conterrà il valore del segno del numero, e ogni byte potrà rappresentare un numero compreso tra `-128` e `127`. I byte hanno un indirizzo che il programma usa per

salvare e recuperare successivamente i dati in maniera performante. Un numero maggiore di `127` o minore di `-128` sarà immagazzinato in più byte adiacenti.

I **tipi interi** sono quattro, e differiscono non per funzionalità ma solo per come vengono immagazzinati in memoria. Il più utilizzato è indubbiamente l'`int` che è a 32 bit. Meno utilizzati ci sono il `byte` (8 bit), lo `short` (16 bit) e il `long` (a 64 bit). I **tipi a virgola mobile** sono due: `float` (a 32 bit) e il più utilizzato `double` (64 bit). Per tutti questi dati esistono varie problematiche (come la **promotion)** per quanto riguardano le operazioni aritmetiche. Il **cast** permette di adattare un certo tipo di dato ad essere trattato come un altro tipo di dato.

Il tipo di **carattere logico-booleano** è il `boolean`, che può assumere come valore solo due literals: `true` o `false`. Un **literal** viene definito come un valore contenente lettere o simboli specificati nel codice sorgente invece che al runtime. L'unico tipo di dato primitivo letterale è il **carattere** `char`, che può essere utilizzato per immagazzinare un singolo carattere incluso tra una coppia di apici singoli. Un carattere è comunque rappresentato da un numero intero ed utilizza lo standard di decodifica Unicode.

Una **stringa** invece può immagazzinare frasi intere e il suo valore va incluso tra due virgolette. Una stringa è un oggetto incluso nel package `java.lang` che ha la caratteristica di essere immutabile e di poter essere istanziato mediante una sintassi breve che non fa uso del costruttore; ciò è molto utile visto che probabilmente è l'oggetto più usato in assoluto nei programmi Java.

Un **reference** è il "nome che diamo ad un oggetto", una particolare variabile che contiene come valore l'indirizzo a cui deve puntare, e che deve conoscere il tipo che andrà a puntare. Quando un oggetto viene istanziato tutte le sue variabili d'istanza vengono automaticamente inizializzate ai propri **valori nulli.**

Il **passaggio di parametri** ad un metodo avviene sempre **per valore**, nel senso che nel momento in cui si passa un valore primitivo ad un parametro di un metodo, verrà copiato nel parametro questo valore. Anche quando si passa un reference verrà passato il suo valore (l'indirizzo del reference) e copiato nel parametro del metodo.

Quando si istanzia un oggetto tutte le variabili d'istanza vengono

inizializzate ai propri valori nulli.

La consultazione della **documentazione della libreria standard** è fondamentale. Possiamo anche creare la documentazione del nostro software con lo stesso stile della documentazione Java, commentando il codice con i commenti di documentazione e utilizzando il comando **javadoc** da riga di comando. Possiamo importare all'interno dei nostri programmi le classi della libreria standard utilizzando il comando **import.**

Un **array** rappresenta un tipo di dato complesso (oggetto) che definisce una collezione di elementi di un certo tipo, indicizzati. Con un solo nome potremo accedere agli elementi della collezione mediante un indice. È possibile creare array di reference, array bidimensionali (non obbligatoriamente matrici) ed array multidimensionali. Il parametro array `String args[]` del metodo `main()`, lo si può utilizzare per passare in **input al programma** alcuni dati provenienti dall'esterno.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Conoscere le regole e le convenzioni per dichiarare identificatori per il codice Java (unità 3.1)	☐	
Comprendere come viene gestita la memoria, i sistemi numerici posizionali, e il metodo del complemento a due (unità 3.2)	☐	
Conoscere e saper utilizzare tutti i tipi di dati primitivi (unità 3.3)	☐	
Saper gestire casting e promotion (unità 3.3)	☐	
Saper utilizzare i reference e capirne la filosofia (unità 3.4)	☐	

Iniziare ad esplorare la documentazione della libreria standard di Java (unità 3.5)	☐	
Saper utilizzare la classe String (unità 3.5)	☐	
Saper commentare il proprio codice ed essere in grado di utilizzare lo strumento javadoc per produrre documentazione tecnica esterna (unità 3.1, 3.5)	☐	
Saper definire, creare, ed inizializzare gli array (unità 3.6)	☐	

Note:

4

Operatori e Gestione del flusso di esecuzione

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Conoscere e saper utilizzare i vari operatori (unità 4.1).
- ✔ Conoscere e saper utilizzare i costrutti di programmazione semplici (unità 4.2).
- ✔ Conoscere e saper utilizzare i costrutti di programmazione avanzati (unità 4.3).
- ✔ Comprendere che prima di iniziare a programmare è necessario prima avere chiari tutti i requisiti, e avere un piano per procedere nella codifica (unità 4.4).

In questo capitolo finalmente affronteremo tutti gli argomenti che ci permetteranno di iniziare a programmare. Parleremo degli operatori e dei costrutti di programmazioni (cicli e condizioni). Infine inizieremo a capire quanto è importante il tipo di approccio che abbiamo alla programmazione, introducendo argomenti importanti come il linguaggio di modellazione UML.

4.1 Operatori di base

Di seguito è presentata una lista completa degli **operatori** che Java mette a disposizione.

> **Java eredita in blocco tutti gli operatori del linguaggio C e quindi, per alcuni di essi, l'utilizzo è alquanto raro.**

4.1.1 Operatore d'assegnazione

L'operatore `=` permette di assegnare un valore ad una variabile. È possibile assegnare valori diversi ad una variabile durante l'esecuzione del codice. Per esempio è lecito effettuare le seguenti assegnazioni:

```
int variabile1 = 1;
int variabile2 = 2;
variabile1 = variabile2;
```

dopo queste istruzioni, il valore intero `2` sarà immagazzinato sia in `variabile1` che in `variabile2`.

4.1.2 Operatori aritmetici

La seguente tabella riassume gli operatori aritmetici semplici definiti dal linguaggio:

Descrizione	**Operatore**
Somma	`+`
Sottrazione	`-`
Moltiplicazione	`*`
Divisione	`/`
Modulo	`%`

L’unico operatore che può risultare non familiare al lettore è l’operatore **modulo.** Il risultato dell’operazione modulo tra due numeri coincide con il resto della divisione fra essi. Per esempio:

```
5 % 3 = 2
10 % 2 = 0
100 % 50 = 0
```

Gli operatori `+` e `−` possono essere usati anche come **operatori unari** (ovvero si possono applicare ad un solo operando) per specificare il segno di un numero. Per esempio le seguenti linee sono perfettamente valide:

```
int i = -1;
int j = +1;
```

Java ha ereditato dalla sintassi del linguaggio C anche altri operatori, sia **binari** (con due operandi) che unari (con un solo operando). Alcuni di essi, oltre a svolgere un’operazione, assegnano anche il valore del risultato ad una variabile utilizzata nell’operazione stessa:

Descrizione	**Operatore**
Somma ed assegnazione	`+=`
Sottrazione ed assegnazione	`-=`
Moltiplicazione ed assegnazione	`*=`
Divisione ed assegnazione	`/=`
Modulo ed assegnazione	`%=`

Per esempio se abbiamo:

```
int i = 5;
```

scrivere:

```
i = i + 2;
```

è equivalente a scrivere:

```
i += 2;
```

4.1.3 Operatori (unari) di pre e post-incremento (e decremento)

La seguente tabella descrive questa singolare tipologia di operatori:

Descrizione	Operatore	Esempio
Pre-incremento di un’unità	`++`	`++i`
Pre-decremento di un’unità	`--`	`--i`
Post-incremento di un’unità	`++`	`i++`
Post-decremento di un’unità	`--`	`i--`

Se vogliamo incrementare di una sola unità una variabile numerica, possiamo scrivere equivalentemente:

```
i = i + 1;
```

oppure:

```
i += 1;
```

ma anche:

```
i++;
```

oppure:

```
++i;
```

ottenendo comunque lo stesso risultato. Infatti, in tutti i casi, otterremo che il valore della variabile `i` è stato incrementato di un’unità ed assegnato nuovamente alla variabile stessa. Quindi anche questi operatori svolgono due compiti (incremento ed assegnazione). Parleremo di **operatore di pre-incremento** nel caso in cui anteponiamo l’operatore d’incremento `++` alla variabile. Parleremo invece di **operatore di post-incremento** nel caso in cui

posponiamo l'operatore di incremento alla variabile. La differenza tra questi due operatori "composti" consiste essenzialmente nelle priorità che essi hanno rispetto all'operatore di assegnazione, quando si trovano all'interno di uno stesso statement. L'operatore di pre-incremento ha maggiore priorità dell'operatore di assegnazione =. L'operatore di post-incremento ha minor priorità rispetto all'operatore di assegnazione =. Ovviamente le stesse regole valgono per gli operatori di decremento.

Facciamo un paio di esempi per rendere evidente la differenza tra i due operatori. Il seguente codice utilizza l'operatore di pre-incremento:

```
x = 5;
y = ++x;
```

Dopo l'esecuzione delle precedenti istruzioni avremo che:

```
x = 6
y = 6
```

Il seguente codice invece utilizza l'operatore di post-incremento:

```
x = 5;
y = x++;
```

In questo caso avremo che:

```
x = 6
y = 5
```

4.1.4 Operatori bitwise

La seguente tabella mostra tutti gli **operatori bitwise** (ovvero che eseguono operazioni direttamente sui bit) definiti in Java:

Descrizione	Operatore
NOT	~
AND	&

OR	`|`
XOR	`^`
Shift a sinistra	`<<`
Shift a destra	`>>`
Shift a destra senza segno	`>>>`
AND e assegnazione	`&=`
OR e assegnazione	`|=`
XOR e assegnazione	`^=`
Shift a sinistra e assegnazione	`<<=`
Shift a destra e assegnazione	`>>=`
Shift a destra senza segno e assegnazione	`>>>=`

Tutti questi operatori binari sono molto efficienti giacché agiscono direttamente sui bit, ma in Java si utilizzano raramente. Infatti in Java non esiste l'aritmetica dei puntatori e, di conseguenza, lo sviluppatore non è abituato a "pensare in bit". L'**operatore NOT** `~` è un operatore unario, dato che si applica ad un solo operando. Per esempio, sapendo che la rappresentazione binaria di `1` è `00000001`, avremo che `~1` varrà `11111110` ovvero `-2`.

Tale operatore, applicato ad un numero intero, capovolgerà la rappresentazione dei suoi bit scambiando tutti gli `0` con `1` e viceversa.

Gli **operatori AND** `&`, **OR** `|` e **XOR** `^`, si applicano a coppie di operandi, e svolgono le relative operazioni logiche di conversioni di bit riassunte nella seguente tabella della verità:

Operando1	Operando2	Op1 AND Op2	Op1 OR Op2	Op1 XOR Op2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Gli **operatori di shift (operatori di scorrimento)** provocano lo scorrimento di un certo numero di bit verso una determinata direzione. Il numero dei bit da scorrere è rappresentato dall'operando a destra dell'operazione. I bit che dopo lo scorrimento si trovano al di fuori della rappresentazione binaria del numero vengono eliminati. I bit che invece "rimangono vuoti" vengono riempiti con i valori `0` oppure `1` a seconda del caso. In particolare, lo scorrimento a sinistra provoca un riempimento con i valori `0` dei bit lasciati vuoti sulla destra della rappresentazione binaria del numero. Anche lo scorrimento a destra senza segno riempie i bit lasciati vuoti con degli `0`. Lo scorrimento a destra con segno, invece, provoca il riempimento di `0` oppure di `1`, a seconda che l'ultima cifra a sinistra prima dello scorrimento (bit del segno) sia `0` oppure `1`, ovvero che la cifra prima dello scorrimento sia positiva o negativa. Consideriamo i seguenti esempi, se abbiamo:

```
byte a = 35; //rappresentazione binaria 00100011
```

e scorriamo (volgarmente "shiftiamo") a destra di due posizioni:

```
a = a >> 2;
```

avremo che:

```
a = 8 // rappresentazione binaria 00001000
```

Se invece abbiamo:

```
byte b = -8; //rappresentazione binaria 11111000
```

e shiftiamo b di una posizione:

```
b = b >> 1;
```

avremo che:

```
b = -4 //rappresentazione binaria 11111100
```

Il seguente è invece un esempio di scorrimento a destra senza segno:

```
int a = -1;
a = a >>> 24;
11111111111111111111111111111111 ovvero -1
>>> 24
00000000000000000000000011111111 ovvero 255
```

Ricordiamo che la promozione automatica delle espressioni avviene per ogni operatore binario e quindi anche per l'operatore di scorrimento a destra senza segno.

L'operazione di scorrimento a destra equivale a dividere l'operando di sinistra per 2 elevato all'operando situato alla destra nell'espressione. Il risultato viene arrotondato per difetto nelle operazioni con resto. Ovvero:

`op1 >> op2` equivale a `op1` diviso (`2` elevato a `op2`).

Similmente, l'operazione di scorrimento a sinistra equivale a moltiplicare l'operando di sinistra per `2` elevato all'operando situato sulla destra dell'operazione. Ovvero:

`op1 << op2` equivale a `op1` moltiplicato (`2` elevato a `op2`).

Vista la velocità d'esecuzione degli operatori di shift conviene usarli in queste situazioni.

4.1.5 Operatori relazionali o di confronto

Il risultato delle operazioni basate su **operatori relazionali** è sempre un valore `boolean`, ovvero `true o false`.

Operatore	Simbolo	Applicabilità
Uguale a	==	Tutti i tipi
Diverso da	!=	Tutti i tipi
Maggiore	>	Solo i tipi numerici
Minore	<	Solo i tipi numerici
Maggiore o uguale	>=	Solo i tipi numerici
Minore o uguale	<=	Solo i tipi numerici

Un classico errore dell'aspirante programmatore è scrivere = in luogo di ==.

Per capire come funzionano leggiamo i seguenti esempi:

```
boolean b = (9 == 9); // b vale true
```

Notiamo che le parentesi tonde sono superflue (anche se aiutano la leggibilità del codice), perché in uno statement l'operatore di assegnazione ha sempre minore priorità rispetto agli altri operatori, quindi verrà eseguito per ultimo.

Se consideriamo il seguente snippet:

```
int i = 10;
double d = 10.0;
boolean b1 = (i != d); // b1 vale false
boolean b2 = b1 == (i >= 11) // b2 vale true
```

la variabile `b1` vale `false` perché `i` non è diverso da `d` (hanno lo stesso valore nonostante siano di tipo diverso). Quindi `i != d` vale `false`, e questo valore viene assegnato a `b1`. Considerando l'ultimo statement, avremo che `b2` invece vale `true`. Infatti `i >= 11` è `false`, come lo è anche `b1` per quanto appena osservato. Quindi l'ultimo statement equivale al seguente:

```
boolean b2 = false == (false)
```

Ma `false == false` vale `true`, e quindi a `b2` verrà assegnato il valore `true`.

Se confrontiamo due reference con l'operatore ==, il risultato risulterà `true` se e solo se i due reference puntano allo stesso oggetto, altrimenti `false`. Infatti viene confrontato sempre il valore delle variabili in gioco. Il valore di una variabile reference, come abbiamo visto nel capitolo precedente, è l'indirizzo in memoria dell'oggetto a cui punta.

Quindi se consideriamo la seguente classe:

```
public class Alunno {
    public String nome;
    public Alunno(String n) {
        nome = n;
    }
}
```

il seguente snippet:

```
Alunno alunno1 = new Alunno("Simone");
Alunno alunno2 = new Alunno("Simone");
System.out.println(alunno1 == alunno2);
```

Stamperà `false`. Infatti il confronto booleano verrà fatto sull'indirizzo dei due oggetti creati, che puntano evidentemente a due oggetti diversi (anche se contengono lo stesso valore per la variabile `nome`).

4.1.6 Operatori logico - booleani

Quelli seguenti sono operatori che utilizzano solo operandi di tipo booleano, che vengono detti **operatori logico - booleani.** Il risultato di un'operazione basata su tali operatori è sempre di tipo `boolean`:

Descrizione	**Operatore**
NOT logico	!
AND logico	&
OR logico	|
XOR logico	^
Short circuit AND	&&

Short circuit OR	`||`
AND e assegnazione	`&=`
OR e assegnazione	`|=`
XOR e assegnazione	`^=`

È facile trovare punti di contatto tra la precedente lista di operatori e la lista degli operatori bitwise. Gli operandi a cui si applicano gli operatori booleani però possono essere solo di tipo booleano.

È consuetudine utilizzare le versioni **short circuit** (letteralmente **corto circuito)** di AND e OR. Ad esempio, la seguente riga di codice mostra come avvantaggiarsi della valutazione logica di corto circuito:

```
boolean flag = ( (a != 0) && (b/a > 10) )
```

Affinché l'espressione tra parentesi sia vera, bisogna che entrambi gli operandi dell'AND siano veri. Se il primo tra loro è in partenza falso, non ha senso andare a controllare la situazione dell'altro operando. In questo caso sarebbe addirittura dannoso perché porterebbe ad una divisione per zero (errore riscontrabile solo al runtime e non in fase di compilazione). Quest'operatore short circuit `&&`, a differenza della sua versione tradizionale `&`, fa evitare il secondo controllo in caso di fallimento del primo. Equivalentemente l'operatore short circuit ||, nel caso la prima espressione da testare risultasse verificata, convalida l'intera espressione senza alcuna altra (superflua) verifica. Per esempio anche con questo codice:

```
boolean flag = ( (a == 0) || (b/a > 10) )
```

la divisione non verrà eseguita.

4.1.7 Concatenazione di stringhe con +

In Java l'operatore `+`, oltre che essere un operatore aritmetico, è anche un operatore per concatenare stringhe. Per esempio il seguente frammento di codice:

```
String nome = "James";
```

```
String cognome = "Gosling";
String nomeCompleto = "Mr. " + nome + " " + cognome;
```

farà in modo che la stringa `nomeCompleto`, abbia come valore `Mr. James Gosling`. Se “sommiamo” un qualsiasi tipo di dato con una stringa, il tipo di dato sarà automaticamente convertito in stringa, e ciò può spesso risultare utile.

4.1.8 Priorità degli operatori

Nella seguente tabella sono riportati, in ordine di priorità, tutti gli operatori di Java. Alcuni di essi non sono ancora stati trattati.

separatori	. [] () ; ,
da sx a dx	`++ -- + - ~ ! (tipi di dati)`
da sx a dx	`* / %`
da sx a dx	`+ -`
da sx a dx	`<< >> >>>`
da sx a dx	`< > <= >= instanceof`
da sx a dx	`== !=`
da sx a dx	`&`
da sx a dx	`^`
da sx a dx	`|`
da sx a dx	`&&`
da sx a dx	`||`
da dx a sx	`?:`
da dx a sx	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`

Fortunatamente non è necessario conoscere a memoria tutte le priorità per

programmare. Nell'incertezza è sempre possibile utilizzare le parentesi tonde così come faremmo nell'aritmetica tradizionale. Non potendo usufruire di parentesi quadre e graffe (poiché in Java queste sono adoperate per altri scopi) sostituiremo il loro utilizzo sempre con parentesi tonde. Quindi, se abbiamo le seguenti istruzioni:

```
int a = 5 + 6 * 2 - 3;
int b = (5 + 6) * (2 - 3);
int c = 5 + (6 * (2 - 3));
```

le variabili `a`, `b` e `c` varranno rispettivamente `14`, `-11` e `-1`.

4.2 Costrutti di programmazione semplici

In ogni linguaggio di programmazione esistono costrutti che permettono al programmatore di controllare la sequenza di esecuzione delle istruzioni. Possiamo dividere questi costrutti in due categorie principali:

- **condizioni** (o **strutture di controllo decisionali**): permettono, durante la fase di runtime, una scelta tra l'esecuzione di istruzioni diverse, a seconda che sia verificata o meno una specificata condizione.
- **Cicli** (o **strutture di controllo iterative**): consentono, in fase di runtime, di decidere il numero di esecuzioni di determinate istruzioni.

In Java le condizioni che si possono utilizzare sono essenzialmente due: il costrutto `if` ed il costrutto `switch`, cui si aggiunge l'operatore ternario. I costrutti di tipo ciclo invece sono quattro: `while`, `for`, `do` (detto anche `do-while`) e il "ciclo for migliorato" (detto anche "foreach").

I costrutti principali (almeno da un punto di vista storico) sono la condizione `if` ed il ciclo `while`. Un programmatore in grado di utilizzare questi due costrutti sarà in grado di codificare un qualsiasi tipo di istruzione. La sintassi di questi due costrutti è alquanto banale e per questo vengono anche detti **costrutti di programmazione semplici.** Nei prossimi due paragrafi studieremo i costrutti `if` e `while`.

4.2.1 Il costrutto `if`

Il **costrutto if** è una condizione che permette di prendere semplici decisioni basate su un'espressione booleana. Un**'espressione booleana** è un'espressione che come risultato può restituire solo valori di tipo `boolean`, vale a dire `true` o `false`. Essa di solito si avvale di operatori di confronto e, se necessario, di operatori logici. La sintassi del costrutto è la seguente:

```
if (espressione-booleana)
    statement;
```

per esempio:

```
if (numeroLati == 3)
    System.out.println("Questo è un triangolo");
```

Nell'esempio, l'istruzione di stampa sarebbe eseguita se e solo se la variabile `numeroLati` avesse valore `3`. In quel caso l'espressione booleana `numeroLati == 3` varrebbe `true` e quindi sarebbe eseguita l'istruzione che segue l'espressione. Se invece l'espressione risultasse `false`, sarebbe eseguita direttamente la prima eventuale statement che segue l'istruzione di stampa. Possiamo anche estendere la potenzialità del costrutto `if` mediante la parola chiave `else`:

```
if (espressione-booleana)
    istruzione1;
else
    istruzione2;
```

per esempio:

```
if (numeroLati == 3)
    System.out.println("Questo è un triangolo");
else
    System.out.println("Questo non è un triangolo");
```

La parola "if" si traduce in italiano con la parola "se"; la parola "else" invece con "altrimenti". Quindi se l'espressione booleana è vera verrà stampata la stringa `Questo è un triangolo`; se è falsa verrà stampata la stringa `Questo non è un triangolo`. Traducendo in italiano il precedente codice si può leggere:

```
Se il numero di lati è uguale a 3
    stampa "Questo è un triangolo"
altrimenti
    stampa "Questo non è un triangolo"
```

che dovrebbe risultare molto chiaro.

Possiamo anche utilizzare blocchi di codice con il seguente tipo di sintassi:

```
if (espressione-booleana) {
    statement_1;
    ...........;
    statement_k;
} else {
    statement_k+1;
    ...........;
    statement_n;
}
```

e questa sintassi che fa uso del blocco di codice è di gran lunga la più utilizzata. Inoltre è obbligatoria utilizzare i blocchi di codice nel caso ci siano più statement da eseguire se una certa espressione booleana è verificata. In definitiva è possibile evitare di mettere le parentesi graffe, solo nel caso ci sia un unico statement da eseguire all'interno del costrutto condizionale.

Possiamo anche comporre più costrutti nel seguente modo:

```
if (espressione-booleana1) {
     statement_1;
     ...........;
     statement_k;
} else if (espressione-booleana2) {
     statement_k+1;
     ...........;
     statement_j;
} else if (espressione-booleana3) {
     statement_j+1;
     ...........;
     statement_h;
} else {
     statement_h+1;
     ...........;
     statement_n;
}
```

Il funzionamento dovrebbe essere abbastanza intuitivo. Se `espressione-booleana1` vale `true` allora verranno eseguite le istruzioni del primo blocco di codice, e di sicuro non verranno eseguite tutte le altre istruzioni in tutti gli altri blocchi di codice. Infatti le altre condizioni booleane verranno ignorate e il flusso di esecuzione del codice si sposterà fuori dall'intero costrutto (dopo l'ultima parentesi graffa che definisce il blocco di codice della clausola `else`). È scontato dire che, se `espressione-booleana1` vale `false` allora si andrebbe a valutare `espressione-booleana2` e se questa fosse `true` si eseguirebbero i relativi statement e si ignorerebbero tutte le altre linee di codice che abbiamo scritto. Ovviamente se anche `espressione-booleana2` valesse `false` allora si andrebbe a valutare `espressione-booleana3` e se questa fosse `true` si eseguirebbero i relativi statement e si ignorerebbe il resto del codice. Infine se tutte e tre le espressioni booleane risultassero non verificate (`ovvero valessero false`) allora verrebbero eseguiti tutti gli statement contenuti nel blocco di codice relativo alla clausola `else`.

Possiamo anche annidare più costrutti. I due seguenti frammenti di codice possono sembrare equivalenti.

```
. . .
if (x != 0)
     if (y < 10)
          z = 5;
     else
          z = 7;
. . .
```

```
. . .
if (x != 0) {
     if (y < 10)
          z = 5;
} else
     z = 7;
. . .
```

In realtà il frammento di codice a sinistra mostra un `if` che annida un costrutto `if - else`. Il frammento di codice a destra invece mostra un costrutto `if – else` che annida un costrutto `if`.

Si consiglia sempre di utilizzare un blocco di codice (mediante l'utilizzo di parentesi graffe) per circondare anche un'unica istruzione. Infatti, questa pratica aggiunge leggibilità. Inoltre capita spesso di aggiungere istruzioni in

un secondo momento.

4.2.2 Il costrutto `while`

Il **ciclo while** permette di eseguire più volte (**iterare**) uno statement (o un insieme di statement compresi in un blocco di codice) tante volte fino a quando una certa condizione booleana è verificata.

La sintassi è la seguente:

```
[inizializzazione;]
while (espressione booleana) {
     statement;
     [aggiornamento iterazione;]
}
```

Come esempio proponiamo una piccola applicazione che stampa i primi dieci numeri:

```
public class WhileDemo {
    public static void main(String args[]) {
        int i = 1;
        while (i <= 10) {
             System.out.println(i);
             i++;
        }
    }
}
```

Analizziamo in sequenza le istruzioni che verrebbero eseguite in fase di runtime (quindi ci limitiamo a considerare le istruzioni all'interno del metodo `main()`). Viene in primo luogo dichiarata ed inizializzata a `1` una variabile intera `i`. Poi inizia il ciclo in cui è esaminato il valore booleano dell'espressione in parentesi. Siccome `i` è uguale ad `1`, `i` è anche minore di `10` e la condizione è verificata. Quindi viene eseguito il blocco di codice nel quale prima sarà stampato il valore della variabile `i` (ovvero `1`) e poi verrà incrementata la variabile stessa di un'unità. Terminato il blocco di codice verrà nuovamente testato il valore dell'espressione booleana. Durante questo secondo tentativo la variabile `i` varrà `2`. Quindi, anche in questo caso, sarà eseguito di nuovo il blocco di codice. Verrà allora stampato il valore della

variabile **i** (ovvero **2)** ed incrementata nuovamente di una unità la variabile stessa. Questo ragionamento si ripete fino a quando la variabile **i** non assume il valore **11.** Quando ciò accadrà, il blocco di codice non verrà eseguito, dal momento che l'espressione booleana non sarà verificata. Il programma quindi eseguirà le istruzioni successive al blocco di codice, ma in questo esempio non ce ne sono, e quindi terminerà.

4.3 Costrutti di programmazione avanzati

In questo paragrafo ci occuperemo di tutti gli altri costrutti di programmazione che regolano il flusso di un'applicazione. È vero che con `if` e `while` possiamo risolvere praticamente tutti i problemi della programmazione, ma è anche vero che a volte sono scomodi da utilizzare rispetto ad altri costrutti. Alcuni di quelli che vedremo in questo paragrafo sono infatti utilizzatissimi. Per esempio il ciclo **for** è molto più utilizzato del ciclo **while.** Altri meno **(break, continue**, operatore ternario) ma sono comunque fondamentali per programmare.

4.3.1 Il costrutto `for`

Il **ciclo for**, è probabilmente il ciclo più completo che offre il linguaggio. Di seguito trovate la sua sintassi nel caso d'utilizzo di una o più istruzioni da iterare.

uno statement:

```
for (inizializzazione; espressione booleana; aggiornamento)
    statement;
```

più statement:

```
for (inizializzazione; espressione booleana; aggiornamento)
{
    statement_1;
    ...........;
    statement_n;
}
```

Il consiglio è sempre quello di utilizzare comunque i blocchi

di codice anche nel caso di statement singolo.

Presentiamo un esempio che stampa i primi 10 numeri partendo da `10` e terminando ad `1`:

```
public class ForDemo {
    public static void main(String args[]) {
        for (int n = 10; n > 0; n--) {
            System.out.println(n);
        }
    }
}
```

In questo caso notiamo che la sintassi è più compatta rispetto a quella relativa al ciclo `while`. Tra le parentesi tonde relative ad un ciclo `for` dichiariamo addirittura una variabile locale `n` (che smetterà di esistere al termine del ciclo). Potevamo anche dichiararla prima del ciclo, nel caso fosse stata nostra intenzione utilizzarla anche fuori da esso.

Per esempio, il seguente codice definisce un metodo che prende come parametro in input un intero `j`, che viene utilizzato nell'espressione booleana. Si può notare come invece la variabile `i` sia dichiarata esternamente al ciclo `for`, visto che viene utilizzata anche dopo di esso per essere stampata:

```
public void forMethod(int j) {
    int i;
    for (i = 0; i < j; ++i) {
        System.out.println(i);
    }
    System.out.println("Numero iterazioni = " + i);
}
```

La sintassi del ciclo `for` è quindi molto flessibile e compatta. Infatti, se nel `while` utilizziamo le parentesi tonde solo per l'espressione booleana, nel `for` le utilizziamo per inserirci rispettivamente prima l'inizializzazione di una variabile, poi l'espressione booleana ed infine l'aggiornamento che sarà eseguito ad ogni iterazione. Si noti che queste tre istruzioni possono anche essere completamente indipendenti tra loro. Potremmo anche dichiarare più variabili all'interno, più aggiornamenti e, sfruttando operatori condizionali, anche più condizioni. Per esempio il seguente codice è valido:

```
public class For {
    public static void main(String args[]) {
         for (int i = 0, j = 10; i < 5 || j > 5; i++, j--) {
             System.out.println("i="+i);
             System.out.println("j="+j);
         }
    }
}
```

Come è possibile notare, le dichiarazioni vanno separate da virgole, ed hanno il vincolo di dover essere tutte dello stesso tipo (in questo caso `int`). Anche gli aggiornamenti vanno separati con virgole, ma non ci sono vincoli in questo caso. Si noti che in questo "settore" del `for` avremmo anche potuto eseguire altre istruzioni, per esempio invocare metodi:

```
for (int i = 0, j = 10; i < 5 || j > 5; i++, j--,
 System.out.println("aggiornamento")) {
     ...
}
```

Questo è il ciclo più utilizzato, vista la sua grande semplicità e versatilità. Inoltre, è l'unico ciclo che permette di dichiarare una variabile con visibilità interna al ciclo stesso.

Il ciclo `while` è utilizzato soprattutto quando non si sa quanto a lungo verranno eseguite le istruzioni e soprattutto nei cicli infiniti, dove la sintassi è banale:

```
while (true) {
    ...
}
```

Segue un ciclo `for` infinito:

```
for (;true;) {
    ...
}
```

che è equivalente a:

```
for (;;) {
    ...
 }
```

4.3.2 Il costrutto do

Il ciclo **do** (detto anche **ciclo do-while)** è il ciclo meno utilizzato. È fondamentale però nel caso in cui si desideri la certezza che le istruzioni in un ciclo siano eseguite almeno nella prima iterazione. Di seguito la sintassi:

```
[inizializzazione;]
do {
    statement;
    [aggiornamento iterazione;]
} while (espressione booleana);
```

In questo caso viene eseguito prima il blocco di codice e poi viene valutata l'espressione booleana (condizione di terminazione) che si trova a destra della parola chiave `while`. Se l'espressione booleana è verificata viene rieseguito il blocco di codice altrimenti esso termina. Si noti il punto e virgola situato alla fine del costrutto. L'output del seguente mini-programma:

```
public class DoWhile {
    public static void main(String args[]) {
        int i = 10;
        do {
           System.out.println(i);
        } while(i < 10);
    }
}
```

è:

```
10
```

Quindi la prima iterazione è stata comunque eseguita.

4.3.3 Ciclo for migliorato

Dalla versione 5 di Java, è stata introdotta una quarta tipologia di ciclo: il cosiddetto **enhanced for loop**, che in italiano potremmo tradurre con **ciclo for migliorato.** L'aggettivo che però più si addice a questo ciclo non è "migliorato" bensì "semplificato". Questo ciclo infatti non è più potente di un tradizionale ciclo `for` e può sostituirlo solo in alcuni casi. In altri linguaggi il ciclo for migliorato viene chiamato **foreach** (in italiano "per ogni") e, per tale

ragione, anche in Java si tende ad utilizzare questo nome, sicuramente più agevole da pronunciare e probabilmente più appropriato. La parola chiave `foreach`, però, non esiste in Java, e per di più Java 8 ha portato un metodo molto importante che si chiama proprio così nel framework delle Collections (che vedremo più avanti). Quindi attenzione a non fare confusione! Viene invece riutilizzata la parola chiave `for`. La sintassi è molto semplice e compatta:

```
for (variabile_temporanea :  oggetto_iterabile) {
    statement;
}
```

dove `oggetto_iterabile` è l'array (o un qualsiasi altro oggetto che vedremo più avanti su cui è possibile iterare) sui cui elementi si vogliono eseguire le iterazioni. Invece `variabile_temporanea` dichiara una variabile a cui, durante l'esecuzione del ciclo, verrà assegnato il valore dell'i-esimo elemento dell'`oggetto_iterabile` all'i-esima iterazione. Quindi `variabile_temporanea` rappresenta all'interno del blocco di codice di questo ciclo un elemento dell'`oggetto_iterabile`; presumibilmente gli statement del costrutto utilizzeranno tale variabile. Notare che non esiste un'espressione booleana per la quale il ciclo deve terminare. Questo è già indicativo del fatto che tale ciclo viene usato soprattutto quando si vuole iterare su tutti gli elementi dell'oggetto iterabile.

Facciamo un esempio:

```
int [] arr = {1,2,3,4,5,6,7,8,9};
for (int tmp :  arr) {
    System.out.println(tmp);
}
```

Il precedente frammento di codice, stampa a video tutti gli elementi dell'array. Il ciclo foreach ha diversi limiti rispetto al ciclo `for` tradizionale. Per esempio non è possibile eseguire cicli all'indietro, e nemmeno è possibile ciclare su più oggetti contemporaneamente. Infine non è possibile accedere all'indice dell'array dell'elemento corrente. In realtà è sempre possibile dichiarare un contatore all'esterno del ciclo e incrementarlo all'interno, ma in casi come questo è forse meglio utilizzare un semplice ciclo `while`.

4.3.4 Il costrutto `switch`

Il **costrutto switch** si presenta come alternativa al costrutto `if`. A differenza di `if` non è possibile utilizzarlo in ogni situazione in cui c'è bisogno di scegliere tra l'esecuzione di parti di codice diverse. Di seguito presentiamo la sintassi:

```
switch (variabile di test)  {
    case valore_1:
        statement_1;
    break;
    case valore_2: {
        statement_2;
        ..........;
        statement_k;
    }
    break;
    case valore_3:
    case valore_4: { //blocco di codice opzionale
        statement_k+1;
        ..........;
        statement_j;
    }
    break;
    [default: {      //clausola default opzionale
        statement_j+1;
        ..........;
        statement_n;
    }]
}
```

A seconda del valore intero che assume la variabile di test vengono eseguite determinate espressioni. La variabile di test deve essere o di tipo stringa, o di un tipo di dato compatibile con un intero, ovvero un `byte`, uno `short`, un `char`, oppure direttamente un `int` (ma non un `long`).

> **Come variabile di test può essere utilizzata anche un'enumerazione o una classe wrapper come `Integer`, `Short`, `Byte` o `Character`. Queste classi sono equiparate ai rispettivi tipi primitivi dalla caratteristica di Java nota come autoboxing-unboxing, che consente di interscambiare**

il tipo primitivo con un oggetto della classe wrapper relativa o viceversa. Tale caratteristica sarà affrontata in dettaglio nel capitolo relativo alle librerie fondamentali. Per quanto riguarda le enumerazioni, queste verranno dettagliate nel capitolo 9, dove verrà anche mostrato come si applica al costrutto `switch`.

Inoltre `valore_1...valore_n` devono essere espressioni costanti e diverse tra loro. Si noti che la parola chiave `break` provoca l'immediata uscita dal costrutto. Se dopo aver eseguito tutte le istruzioni che seguono un'istruzione di tipo `case` non è presente un'istruzione `break`, verranno eseguiti tutti gli statement che seguono gli altri `case`, sino a quando non si arriverà ad un `break` (questa tecnica è nota come **fallthrough).** Di seguito viene presentato un esempio:

```
public class SwitchStagione {
    public static void main(String args[]) {
    int mese = 4;
    String stagione;
    switch (mese) {
         case 12:
         case 1:
         case 2:
              stagione = "inverno";
         break;
         case 3:
         case 4:
         case 5:
              stagione = "primavera";
         break; //senza questo break si ha estate
         case 6:
         case 7:
         case 8:
              stagione = "estate";
         break;
         case 9:
         case 10:
         case 11:
         stagione = "autunno";
         break;
```

```
            default: //la clausola default è opzionale
                 stagione = "non identificabile";
            }
            System.out.println("La stagione e' " + stagione);
        }
    }
```

Segue un altro esempio che fa uso di una stringa come variabile di test.

```
    public String getTipoGiornoSettimana(String
    giornoDellaSettimana) {
         String typeOfDay;
         switch (giornoDellaSettimana) {
             case "Monday":
                 typeOfDay = "Inizio settimana";
                 break;
             case "Tuesday":
             case "Wednesday":
             case "Thursday":
                 typeOfDay = "Settimana piena";
                 break;
             case "Friday":
                 typeOfDay = "Fine settimana lavorativa";
                 break;
            case "Saturday":
            case "Sunday":
                 typeOfDay = "Weekend";
                 break;
            default:
                 typeOfDay = "Indefinito!";
                 break;
        }
        return typeOfDay;
    }
```

Si ricorda che le stringhe sono case sensitive e bisogna stare attenti. In generale è buona norma evitare il fallthrough (per la sua scarsa leggibilità) ed usare il costrutto `switch` nella maniera più semplice possibile. È sempre consigliabile usare una clausola `default` anche quando sembra non essercene bisogno. Infatti a priori non sappiamo se il nostro programma si evolverà ampliando i possibili `case` da definire in uno `switch`. La clausola `default` potrebbe servire sia per gestire i nuovi `case`, con un comportamento standard, sia per scoprire che il costrutto `switch` deve essere modificato per

accomodare i nuovi `case`. La clausola `default` nel costrutto `switch` è l'equivalente della clausola `else` nel costrutto `if`.
Infine è sempre consigliabile mantenere un ordine logico dei vari `case` per non incorrere in dimenticanze e peggiorare la leggibilità.

> **Se state utilizzando come editor EJE, potete sfruttare scorciatoie per creare i cinque principali costrutti di programmazione. Potete infatti sfruttare il menu Inserisci (o Insert se avete scelto la lingua inglese) o le eventuali scorciatoie con la tastiera (CTRL-2, CTRL-3, CTRL-4, CTRL-5, CTRL-6, etc.). In particolare è anche possibile selezionare una parte di codice per poi circondarla con un costrutto.**

4.3.5 L'operatore ternario

Esiste un operatore non ancora trattato fin qui, che qualche volta può sostituire il costrutto `if`. Si tratta del cosiddetto **operatore ternario** (detto anche **operatore condizionale)**, che può regolare il flusso di esecuzione come una condizione. Di seguito si può leggerne la sintassi:

```
variabile = (espressione-booleana) ? espr1 :  espr2;
```

dove se il valore della `espressione-booleana` è `true` si assegna a `variabile` il valore di `espr1;` altrimenti si assegna a `variabile` il valore di `espr2`.

Requisito indispensabile è che il tipo della `variabile` e quello restituito da `espr1` ed `espr2` siano compatibili. È escluso il tipo `void`.

L'operatore ternario non può essere considerato un sostituto del costrutto `if`, ma è molto comodo in alcune situazioni. Un esempio pratico per chi ha già avuto a che fare con un database è il seguente:

```
String query = "select * from table " +
    (condition != null ? "where " + condition :  "");
```

Abbiamo creato una stringa contenente una query SQL e, se la stringa `condition` è diversa da `null`, aggiunge alla query la condizione, altrimenti

aggiunge una stringa vuota.

4.3.6 `break` e `continue`

La parola chiave **break** è stata appena presentata come comando capace di fare terminare il costrutto `switch`. Ma `break` è utilizzabile anche per far terminare un qualsiasi ciclo. Il codice seguente mostra un modo alternativo per stampare i primi dieci numeri interi:

```
int i = 0;
while (true) { //ciclo infinito
    if (i > 10) {
        break;
    }
    System.out.println(i);
    i++;
}
```

In questo esempio, una volta verificata l'espressione booleana `i > 10`, il ciclo viene interrotto dalla parola chiave `break` e il flusso di esecuzione continua dopo la fine del ciclo.
Oltre al `break` esiste la parola chiave **continue**, che fa terminare non l'intero ciclo, ma solo l'iterazione corrente (ovvero fa saltare alla prossima iterazione).
Il seguente frammento di codice provoca la stampa dei primi dieci numeri, escluso il cinque:

```
int i = 0;
do {
    i++;
    if (i == 5) {
        continue;
    }
    System.out.println(i);
} while(i <= 10);
```

Sia `break` sia `continue` possono utilizzare etichette (label) per specificare, solo nel caso di cicli annidati, su quale ciclo devono essere applicati. Il seguente frammento di codice stampa una sola volta i soliti primi dieci numeri interi:

```
int j = 1;
pippo: //possiamo dare un qualsiasi nome ad una label
while (true) {
    while (true) {
        if (j > 10)
            break pippo;
        System.out.println(j);
        j++;
   }
}
```

Una label ha quindi la seguente sintassi:

```
nomeLabel:
```

Una label può essere posizionata solo prima di un blocco di codice che di solito è definito da un ciclo (come nell'esempio precedente); ma potremmo anche posizionare una label al di fuori di un `if` o di un blocco di codice qualsiasi come nel seguente esempio:

```
int j = 1;
labelPerBloccoDiCodice: //possiamo dare un qualsiasi nome ad
una label
{
    while (true) {
        if (j > 10)
            break labelPerBloccoDiCodice;
        System.out.println(j);
        j++;
    }
}
```

Nell'esempio precedente non potremmo usare il `continue` visto che la label non è relativa a un ciclo. Inoltre è chiaro che il `break` e il `continue` usati con label, devono trovarsi all'interno del blocco di codice "etichettato".

In generale l'utilizzo di break e continue non è incoraggiato, e l'utilizzo delle label addirittura sconsigliato. Tuttavia capitano delle situazioni di programmazione nelle quali questi strumenti rappresentano soluzioni accettabili.

4.4 Programmare meglio

I cicli e le condizioni sono solo gli strumenti che ci permetteranno di gestire il flusso di controllo delle nostre applicazioni. Padroneggiare questi strumenti è fondamentale ma non basta. Sarà il modo in cui approcciamo alla programmazione che sarà determinante.

4.4.1 Approccio alla programmazione

Quando è il momento di iniziare un nuovo programma, l'istinto del programmatore è quello di scrivere il codice al più presto. Può sembrare giusto, ma come abbiamo già accennato nel secondo capitolo, *programmare* significa eseguire un processo come questo:

1. capire bene cosa si deve fare;
2. decidere come farlo;
3. implementare la soluzione;
4. testare il programma e con tutta probabilità tornare sul codice per risolvere eventuali problemi o apportare migliorie.

Il primo passo deve essere obbligatoriamente quello di capire tutte le funzionalità che il software deve implementare, ovvero capire cosa fare. Questo può sembrare ovvio, ma vedremo che invece spesso non lo è affatto. Per definire tutte le funzionalità che un programma deve implementare si possono utilizzare delle tecniche specifiche, altrimenti rischieremo di non avere i prerequisiti per procedere correttamente. Nel capitolo 6 approfondiremo questo argomento quando parleremo di Analisi Object Oriented.

Il secondo punto consiste nel decidere come implementare il codice (prima di scriverlo). Tantissimi programmatori però preferiscono saltare il secondo punto e passare direttamente al terzo. La pratica di "buttarsi direttamente sul codice" può essere eseguita correttamente quando si ha la necessaria esperienza, ma è più difficile quando questa manca. Quando programmiamo in Java infatti, scriveremo classi da cui istanzieremo oggetti che risolveranno i problemi da risolvere al runtime. Gli oggetti collaboreranno tra loro chiamando metodi, quindi eseguendo azioni. Messa così non sembra tanto semplice scrivere un programma Java, e in effetti è

proprio così: non è semplice! Benché la teoria dell'object orientation risulti chiara e da subito familiare perché ispirata al mondo reale, la pratica invece potrebbe rivelarsi piuttosto ostica, a seconda della forma mentis del programmatore. A partire dal capitolo 6 approfondiremo anche questi argomenti quando parleremo di Progettazione Object Oriented. Intanto per poter approcciare correttamente al problema eseguendo anche il punto due del processo consigliato sopra, introduciamo in maniera semplificata ed informale, una definizione fondamentale per la programmazione: l'algoritmo.

4.4.2 Algoritmo

Un **algoritmo** è una successione di istruzioni o passi che definiscono le operazioni da eseguire su dei **dati** per risolve un determinato **problema.**

Molte persone identificano la programmazione stessa con il concetto di algoritmo. In effetti una delle attività cardine della programmazione consiste proprio in questo, risolvere problemi eseguendo dei passi, anche se in realtà vedremo che la programmazione moderna non è fatta solo di algoritmi. Il concetto di algoritmo non è esclusivo del campo informatico e può essere applicato ad ogni campo, anche nella vita di tutti i giorni. Un esempio reale di algoritmo sono i libretti di istruzioni di montaggio di un mobile. Con una serie di istruzioni da eseguire in sequenza, essi ci permettono di realizzare un mobile utilizzando oggetti vari (viti, assi di legno di diversa misura, vari attrezzi di lavoro, etc.). In questo caso il problema è "costruire il mobile", i dati sono i materiali che abbiamo a disposizione, l'algoritmo è rappresentato dal libretto d'istruzioni. Una volta definito l'algoritmo, risolvere il problema significa solo eseguire le istruzioni riportate in esso. Senza libretto d'istruzioni, e senza l'adeguata esperienza, non sarà facile montare il mobile, esattamente come nella programmazione non sarà facile creare un programma senza prima decidere quali attività eseguire.
Nell'esempio del paragrafo 4.3.1, avevamo scritto:

```
if (numeroLati == 3)
    System.out.println("Questo è un triangolo");
else
    System.out.println("Questo non è un triangolo");
```

e avevamo detto che potevamo tradurre questo codice in italiano in questo

modo:

```
Se il numero di lati è uguale a 3
    stampa "Questo è un triangolo"
altrimenti
    stampa "Questo non è un triangolo"
```

Si può notare che se avessimo scritto un algoritmo per l'esempio in questione, questo non sarebbe stato molto differente da questa traduzione.

Continuiamo con un altro esempio: l'algoritmo scritto in linguaggio naturale per "preparare una tazza di tè":

- Dati di input: tazza, acqua, bollitore elettrico, tè (varie tipologie)
    1. riempire la tazza d'acqua sino al livello desiderato
    2. versare l'acqua dalla tazza nel bollitore elettrico
    3. accendere il bollitore elettrico
    4. scegliere la tipologia di tè
    5. posare la bustina di tè nella tazza
    6. quando il bollitore elettrico si spegne, versare il contenuto di acqua calda nella tazza
    7. aspettare i minuti necessari per la corretta infusione e rimuovere la bustina dalla tazza
    8. aggiungere zucchero se necessario
- Output: tazza di tè pronta

Questo è solo uno degli algoritmi possibili per preparare una tazza di tè. È probabile che nel preparare il tè direttamente senza algoritmi, i punti 1 e 2 sarebbero sostituiti con il seguente:

1. Riempire il bollitore elettrico con una quantità d'acqua quantomeno sufficiente a riempire la tazza come desiderato

Anche i punti 4 e 5 forse sarebbero stati anticipati o forse posticipati. Con questo algoritmo invece, si è cercato di ottimizzare il livello dell'acqua desiderata ed eliminare gli sprechi con i punti 1 e 2. Inoltre abbiamo provato

ad ottimizzare anche i tempi morti inserendo i punti 4 e 5 mentre aspettiamo che il bollitore faccia il proprio dovere. Questo dovrebbe far pensare al fatto che la formalizzazione di un algoritmo, dà la possibilità di pensare a come ottimizzare le attività da eseguire. Se invece si prepara direttamente il tè senza avere un algoritmo da seguire, probabilmente ci si dovrebbe concentrare sulla pratica (prendere la tazza senza farla cadere, fare attenzione a non scottarsi con l'acqua calda etc.) e non si avrebbe il tempo e la concentrazione per ottimizzare la preparazione. È questo il punto fondamentale, creare un piano prima di eseguirlo. Anche quando si programma, il programmatore dedica parte delle sue risorse mentali ad altre cose correlate alla programmazione, come l'usare correttamente la tastiera, interagire con l'ambiente di sviluppo nella maniera più proficua, documentarsi sulle librerie da utilizzare e così via. Quindi conviene partire con le idee chiare.

Per scrivere un algoritmo si possono usare vari strumenti, il linguaggio naturale, un linguaggio di programmazione, dello pseudocodice o anche un linguaggio di notazione. Scrivere un algoritmo in linguaggio naturale (ovvero in italiano o volendo in un'altra lingua) sarà probabilmente la vostra prima scelta inizialmente (vedi esempio della preparazione della tazza di tè). Scrivere un algoritmo direttamente in un linguaggio di programmazione invece, è una scelta spesso troppo complessa, soprattutto quando si inizia. Probabilmente utilizzare lo **pseudocodice**, ovvero creare l'algoritmo spiegando alcune parti con il linguaggio naturale ed altre con il linguaggio di programmazione, rappresenta il giusto compromesso. Infine si potrebbe utilizzare un **linguaggio di modellazione** come l'**UML** per rappresentare graficamente un algoritmo.

4.4.3 Introduzione ad UML

L'**UML**, acronimo di **Unified Modeling Language** (letteralmente "linguaggio di modellazione unificato") è un linguaggio i cui elementi costitutivi sono per lo più grafici elementari come rettangoli, ovali, omini stilizzati, frecce e così via. Questi elementi vengono utilizzati all'interno di diagrammi seguendo certe regole. È un linguaggio basato sul paradigma orientato agli oggetti, e mette a disposizione diverse tipologie di diagrammi che possono essere utilizzati in situazioni e per scopi diversi.

Più formalmente possiamo definire UML come un linguaggio che permette ad un sistema software di:

1. creare specifiche: può aiutare a definire cosa deve fare il sistema.
2. Costruire: può aiutare a capire come deve essere fatto il sistema.
3. Visualizzare: consente di visualizzare il software da altri punti di vista diversi dal codice.
4. Documentare: con i diagrammi UML possiamo spiegare il nostro software senza leggerne il codice.

La definizione di UML è molto astratta e complicata, quindi cercheremo di capirla direttamente utilizzandolo all'interno di questo libro per i nostri esempi. Nell'appendice I trovate comunque altre informazioni interessanti su UML.

In breve possiamo dire che UML si può utilizzare in tanti modi diversi. Con UML si possono creare **specifiche di implementazione**, ovvero creare documenti talmente completi e precisi che dati nelle mani di un programmatore verranno trasformati in codice funzionante in breve tempo. Una situazione purtroppo rara, ma tutto sommato fattibile. Addirittura l'obiettivo principale iniziale di UML era quello di diventare un **linguaggio eseguibile:** progettare soluzioni dettagliatissime senza essere distratti da un linguaggio di programmazione, e poi usare tool che trasformano i nostri diagrammi in codice funzionante. Una promessa realizzata solo in parte nella realtà, e che somiglia ormai molto di più ad un'utopia.

In questo libro invece ci limiteremo ad utilizzare **UML con pochi dettagli**, quel tanto che ci serve per chiarirci le idee, per abbozzare soluzioni, per anticipare l'individuazione di errori, ma senza la pretesa di risolvere problemi di programmazione senza programmare. Utilizzeremo UML per renderlo funzionale ai nostri scopi, evitando di cadere nella trappola della saccenza fine a sé stessa. Ovvero, nonostante la conoscenza di elementi di sintassi avanzata entusiasmanti, non ne forzeremo l'utilizzo. Per tale ragione non useremo tool specifici per disegnare i nostri diagrammi, e sentiamoci liberi di disegnare su semplici fogli di carta quando serve.

UML definisce diversi diagrammi (vedremo solo i più utilizzati) all'interno dei quali vengono utilizzati elementi grafici elementari. Possiamo

dividerli in due tipologie: i diagrammi statici e i diagrammi dinamici.

I **diagrammi statici**, il cui insieme viene definito il **modello statico** del sistema, rappresentano gli aspetti statici del sistema software. Appartengono a questa tipologia il **diagramma dei casi d'uso (use case diagram)**, il **diagramma delle classi (class diagram)**, il **diagramma di installazione (deployment diagram)** e il **diagramma dei componenti (component diagram).**

I diagrammi dinamici, il cui insieme viene definito modello dinamico, rappresentano i comportamenti del sistema software. Appartengono a questa tipologia il **diagramma di sequenza (sequence diagram)**, il **diagramma di collaborazione** (**collaboration diagram**), il **diagramma di stato (state diagram)** e il **diagramma delle attività (activity diagram).**

Per esempio un algoritmo, che descrive una funzionalità del sistema, può essere rappresentato proprio con un diagramma di attività UML. Per esempio nella figura 4.1, è rappresentato un algoritmo semplificato che mostra come suonare una scala con uno strumento musicale leggendo da uno spartito.

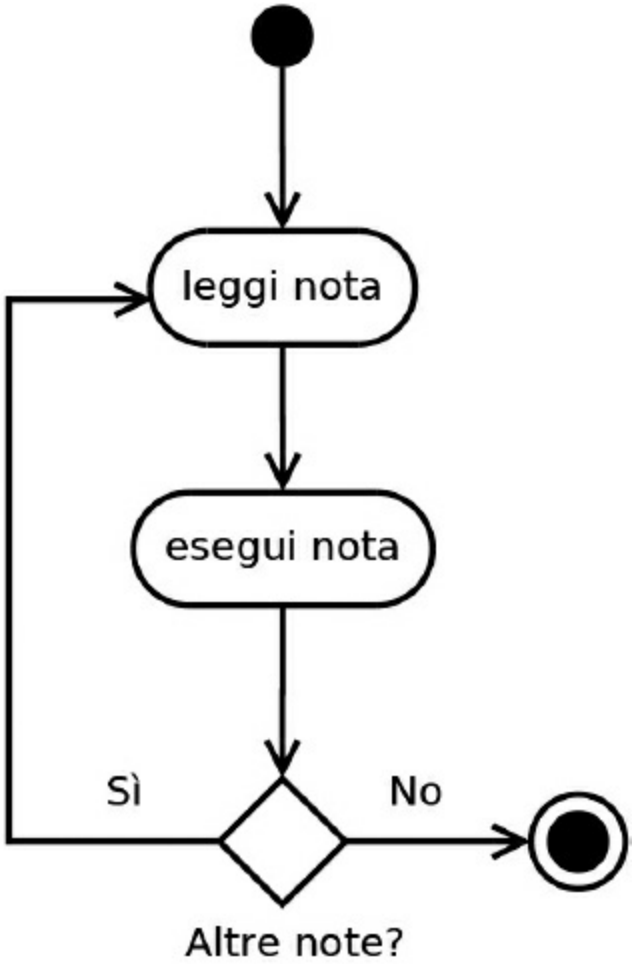


Figura 4.1 - Activity Diagram “Suonare una scala”.

La notazione è molto semplice ed intuitiva. Il cerchio nero, rappresenta il punto di inizio dell’attività, il cerchio bianco con all’interno il cerchio nero concentrico, rappresenta la fine dell’attività. Gli ovali rappresentano le attività vere e proprie, il rombo una condizione (in questo caso la condizione di un ciclo), le frecce le transizioni che portano da un elemento del diagramma ad un altro.

L’appendice L è interamente dedicata alla sintassi base di UML. In questo esempio abbiamo semplificato la sintassi.

Riscrivendo l’algoritmo in linguaggio naturale:

1. leggi la nota
2. esegui la nota
3. se ci sono altre note sullo spartito torna al punto 1

Possiamo utilizzare (ma non è obbligatorio) un activity diagram per descrivere i nostri algoritmi. In qualsiasi caso impareremo ad utilizzarli negli esempi e negli esercizi di questo libro.

Riepilogo

L'**operatore d'assegnazione** `=` permette di assegnare un valore ad una variabile. Gli **operatori aritmetici** `+`, `-`, `*` (per) e `/` (diviso), funzionano esattamente come nella aritmetica tradizionale, l'operatore modulo `%` invece, restituisce il resto di una divisione. Il simbolo `+` viene usato anche come **concatenatore di stringhe.**

Gli **operatori pre e post-incremento** `++` e `--` permettono di incrementare o decrementare di un'unità un numero. Differiscono tra loro solo per quanto riguarda la priorità all'interno di un'espressione che contiene un'assegnazione, altrimenti pre o post-incremento funzionano esattamente allo stesso modo.

Gli **operatori bitwise** (`~`, `&`, `|`, `^`, `<<`, `>>`, `>>>`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`) consentono di modificare direttamente la rappresentazione dei bit degli operatori in modo da ottenere prestazioni molto alte, ma vengono usati raramente.

Il risultato delle operazioni basate su **operatori relazionali** (`==`, `!=`, `<`, `>`, `<=`, `>=`) è sempre un valore `boolean`.

Gli operandi e il risultato di un'operazione basata su **operatori logico - booleani** (`!`, `&`, `|`, `^`, `&&`, `||`, `&=`, `|=`, `^=`) è di tipo `boolean`. Gli operatori di corto circuito sono più usati perché più potenti ed efficienti. Essi permettono di evitare di valutare il secondo operando nel caso non sia necessario.

Tutti gli operatori presenti all'interno di uno statement, sono eseguiti tramite una **tabella delle priorità.** È possibile forzare le priorità mediante l'uso di parentesi tonde.

Per gestire il flusso di esecuzione abbiamo bisogno di **condizioni e cicli.**

Per quanto concerne i cicli, di sicuro il più utilizzato è il **costrutto if.**

Insieme alla sua **clausola else** consente di direzionare il flusso di esecuzione dell'applicazione. In base a una o più condizioni può essere eseguita una certa parte di codice piuttosto che un'altra. Stesso discorso vale per il **costrutto switch**, che però meglio si adatta nelle situazioni in cui le condizioni da verificare si basano sull'operatore di confronto. L**'operatore ternario** invece permette di creare condizioni al volo molto dinamicamente.

Il **for** è il più completo dei cicli, consentendo anche la definizione di una o più variabili locali all'interno della sua stessa definizione. Il **ciclo while** è più adatto per cicli più semplici, per esempio i cicli infiniti. Il **ciclo "for migliorato"** invece è il più pratico e meno verboso (ovvero più sintetico) ma non permette di utilizzare indici al suo interno. Il **ciclo do** invece è consigliato quando si vuole che almeno una volta il suo blocco di codice sia eseguito, visto che la condizione viene valutata solo dopo la prima iterazione.

La parola chiave **break**, permette di uscire dal ciclo in cui si trova, mentre la parola chiave **continue**, consente di saltare fuori dall'iterazione corrente e proseguire con la successiva. Entrambe queste parole chiave possono usufruire di **label** per saltare anche al di fuori del ciclo dove sono dichiarate (comportamento sconsigliato).

Per chi inizia, è importante sapere che prima di scrivere il codice bisogna capire **cosa** implementare, e **come** implementarlo. L**'algoritmo** definisce una serie di passi che possono essere eseguiti per realizzare il software. È possibile scriverlo con linguaggio naturale, linguaggio di programmazione, pseudocodice o un linguaggio di modellazione.

L'**UML** è lo standard de facto dei linguaggi di modellazione; definisce diagrammi nei quali è possibile utilizzare semplici elementi grafici, per costruire, creare specifiche, visualizzare e documentare un sistema software.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In

		Data
Conoscere e saper utilizzare i vari operatori (unità 4.1)	☐	
Conoscere e saper utilizzare i costrutti di programmazione semplici (unità 4.2)	☐	
Conoscere e saper utilizzare i costrutti di programmazione avanzati (unità 4.3)	☐	
Comprendere che prima di iniziare a programmare è necessario prima avere chiari tutti i requisiti, e avere un piano per procedere nella codifica (unità 4.4)	☐	

Note:

5

Sviluppare realmente con Java

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Saper interagire con JShell per sperimentare, esplorare e scoprire Java (unità 5.1).
- ✔ Avere una conoscenza delle potenzialità degli IDE (unità 5.2).
- ✔ Comprendere le difficoltà di lavorare in una realtà aziendale (unità 5.3).
- ✔ Saper utilizzare un processo che permetta di iniziare un programma da zero (unità 5.4).

Lo scopo principale e fondamentale di questo capitolo è far capire cosa significa sviluppare in Java nella realtà, oltre la programmazione. Nella prima parte viene introdotto JShell, un software che dovrebbe favorire l'apprendimento del linguaggio, permettendoci di eseguire velocemente frammenti di codice fuori di ogni contesto. Nella seconda parte del capitolo invece, introdurremo brevemente cosa sono gli IDE e quali sono i vantaggi nell'utilizzarli. Non scenderemo nei dettagli, cercheremo solo di dare un'idea di quali sono le potenzialità di un IDE. Inoltre introdurremo alcuni aspetti con cui si ha a che fare in una realtà aziendale. Questo dovrebbe servire a prepararsi per affrontare il mondo del lavoro senza troppi traumi. L'ultima

parte del capitolo è la più importante. In modo pratico cercheremo di definire un processo per creare un software da zero.

5.1 JShell

Una delle novità più importanti che ha portato con sé Java 9, non riguarda il linguaggio, ma il JDK. Infatti è stato aggiunto alla suite di applicazioni storiche come java, javadoc, javac, etc., la console **JShell.**

Si tratta di un software della tipologia **REPL**, acronimo di **Read Evaluate Print Loop**, che possiamo tradurre come ciclo leggi-valuta-stampa. Infatti un software REPL è implementato di solito come una console interattiva, che quando attivata apre un loop (ciclo) con cui si mette in attesa di input da parte dell'utente. Ad ogni input che viene confermato dall'utente il software lo legge, lo valuta e stampa un risultato. Per testare come funziona una certa parte di codice, solitamente dobbiamo scrivere un programma intero, che come minimo richieda la compilazione e l'esecuzione di una classe che contenga il metodo `main()`. Poi bisogna risolvere eventuali errori ripetendo il ciclo scrivi-compila-esegui-valuta, tante volte fino a quando non risolveremo ogni problema.

Con JShell è possibile scrivere vari tipi di **snippet** come statement, espressioni, metodi, variabili, classi, import e così via, senza dover obbligatoriamente scrivere le espressioni nei metodi, o i metodi nelle classi o scrivere un metodo `main()`. Non è nemmeno necessario compilare: ogni input passato a JShell viene immediatamente letto e valutato, quindi viene stampato un messaggio di risposta sulla valutazione. Non ci sarà neanche bisogno di usare obbligatoriamente i comandi di stampa come abbiamo fatto sinora. JShell ci permetterà di risparmiare tempo e ci invoglierà a sperimentare più facilmente il nostro codice, a creare prototipi, scoprire le librerie standard come funzionano davvero ed altro ancora. Può quindi accelerare il processo di apprendimento.

JShell non sostituisce un IDE, ma è possibile copiare il codice da un IDE e incollarlo in JShell per semplificarne il test. Naturalmente è possibile anche copiare il codice da JShell per incollarlo all'interno di un IDE.

5.1.1 HelloWorld con JShell

Per **aprire una sessione** con JShell, è necessario semplicemente aprire una prompt dei comandi (programma cmd), e da lì eseguire il comando:

```
jshell
```

(è possibile anche eseguire direttamente il comando jshell dal menu Start di Windows, senza aprire prima la prompt dei comandi).

JShell ci darà il benvenuto e ci inviterà ad eseguire un comando per ottenere ulteriori informazioni. Digitando quindi il comando suggerito:

```
/help intro
```

JShell ci descriverà come funziona in breve, mostrando anche qualche semplice esempio, come è possibile osservare nella figura 5.1.

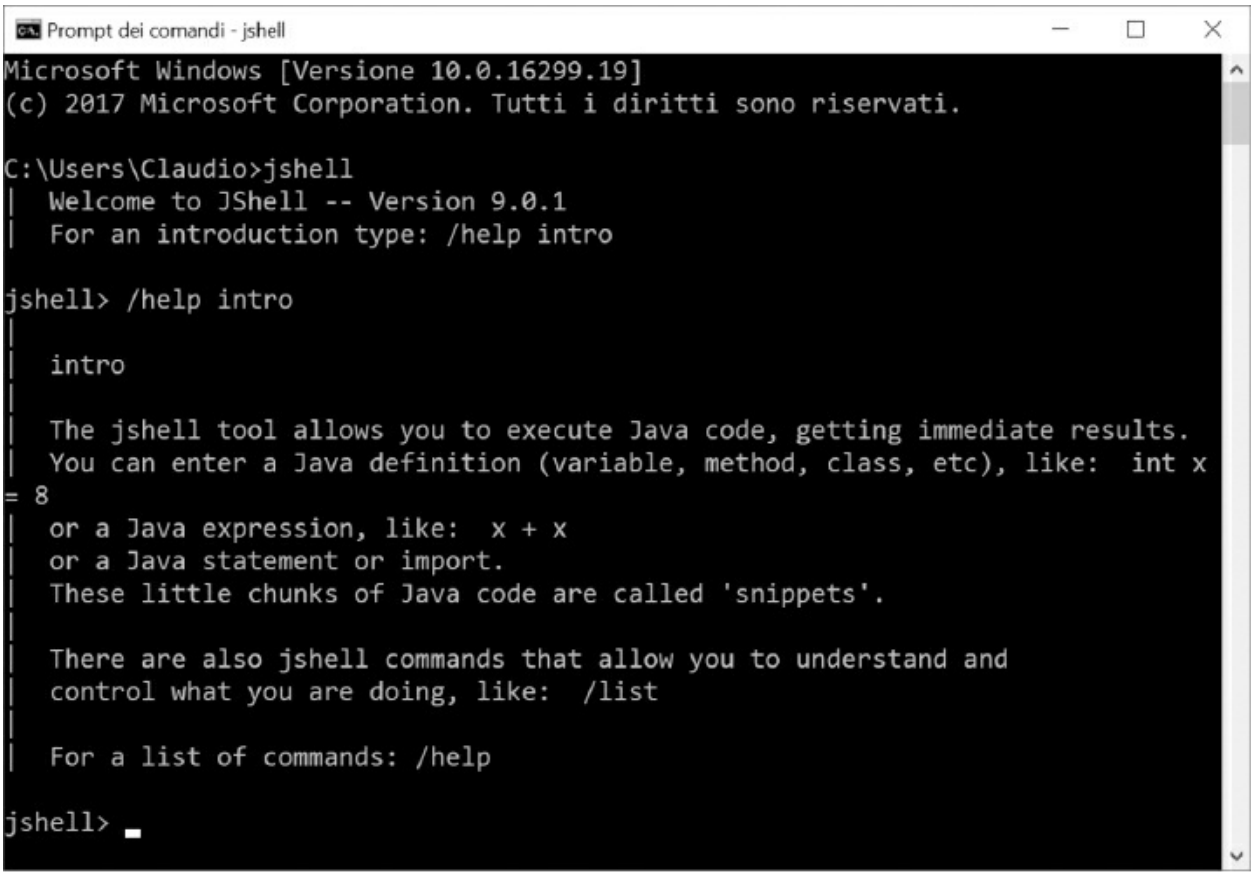


Figura 5.1 - Apertura sessione JShell ed esecuzione comando "/help intro".

Nel messaggio di introduzione viene subito fatta una distinzione tra i

comandi di JShell (per esempio /help intro) e i cosiddetti **snippet** (ovvero i frammenti di codice Java che JShell legge, valuta e stampa). I comandi JShell iniziano tutti con il simbolo /, e possiamo leggerne la lista completa digitando semplicemente il comando:

```
/help
```

oppure più brevemente:

```
/?
```

Invece possiamo scrivere qualsiasi istruzione Java all'interno di una sessione JShell, e lasciarla processare premendo il tasto Enter (Invio). Per esempio possiamo stampare al volo la stringa `Hello World!` nel seguente modo:

```
jshell> System.out.println("Hello World!")
```

ed ottenere la risposta:

```
Hello World!
```

> **Si noti che non c'è stato neanche bisogno di utilizzare il; finale, che ha caratterizzato tutti gli statement scritti sinora. Questo perché JShell, come vedremo nel prossimo paragrafo, ha regole meno restrittive rispetto al classico compilatore Java.**

Se digitassimo il nostro codice scorrettamente:

```
jshell> System.out.PRINTLN("Hello World!")
```

otterremo un errore da JShell:

```
|  Error:
|  cannot find symbol
|    symbol: method PRINTLN(java.lang.String)
|  System.out.PRINTLN("Hello World!")
|  ^-----------------^
```

Per uscire da una sessione JShell (ed essere salutati), basta digitare il

comando:

```
jshell> /exit
|  Goodbye
```

5.1.2 JShell e codice Java

Abbiamo visto che all'interno di una sessione JShell è possibile alternare **codice Java** (snippet, classi, metodi, e così via) e **comandi JShell.** Per quanto riguarda il codice Java, JShell si pone l'obiettivo di rappresentare uno strumento per testarlo nel modo più veloce possibile. Come abbiamo già notato non occorre creare per forza classi, metodi e seguire tutte le classiche regole di Java, bensì tale strumento cerca di facilitare la codifica e quindi ci sono alcune regole un po' diverse rispetto a quelle classiche. Partiamo subito con degli esempi.

> **È consigliato al lettore di scrivere le istruzioni seguenti passo dopo passo per familiarizzare con JShell. Tra un esempio e l'altro potrà anche testare altre istruzioni. Si può anche tranquillamente lavorare con due sessioni JShell separate.**

5.1.2.1 Esempi pratici

La **prima regola** che cambia l'abbiamo già notata: con le espressioni su di un'unica riga non c'è bisogno di utilizzare il `;` finale. Possiamo quindi dichiarare una variabile intera `uno`, inizializzata con il valore `1` nel seguente modo:

```
jshell> int uno = 1
uno ==> 1
```

Si noti come JShell abbia restituito la valutazione dello snippet stampando a video:

```
uno ==> 1
```

che va interpretato come "la variabile `uno` vale `1`". Ora dichiariamo una

variabile booleana senza inizializzarla:

```
jshell> boolean b
b ==> false
```

In questo caso la variabile booleana `b` è inizializzata automaticamente al suo valore di default `false` automaticamente (come una variabile d'istanza quando viene istanziato un oggetto). Abbiamo quindi già trovato la **seconda regola** differente rispetto al compilatore Java, che riguarda l'inizializzazione delle variabili. Le variabili degli snippet non inizializzate sono automaticamente impostate ai propri valori nulli, come se fossero variabili d'istanza. Se proviamo a dichiarare un'altra variabile con lo stesso nome della precedente (`b`) nel seguente modo:

```
jshell> byte b
b ==> 0
```

JShell non avrà problemi ad utilizzarla. Infatti la nuova dichiarazione della variabile `b` va a sovrascrivere la precedente dichiarazione. All'interno di codice Java ordinario, questa situazione avrebbe provocato un errore in compilazione, perché non si possono dichiarare due variabili con lo stesso nome. Quindi ecco una **terza regola** differente per JShell rispetto al compilatore Java: dichiarare due volte la stessa variabile (o lo stesso metodo) causa la sovrascrittura della prima variabile dichiarata (o del primo metodo dichiarato). Nella programmazione standard questa situazione provocherebbe un errore in compilazione. Anche in questo caso non avendo fornito un'inizializzazione alla variabile `b`, viene assegnato automaticamente il valore nullo per il tipo di dato, che per un tipo `byte` equivale a `0`. Assegniamo poi il valore `128` alla variabile `b`. Ricordiamo che il valore massimo che può assumere un tipo di dato byte è `127`, quindi otterremo un errore:

```
jshell> b = 128
|  Error:
|  incompatible types: possible lossy conversion from int to
byte
|  b = 128
|      ^-^
```

I messaggi di JShell seguono sempre un simbolo di "|" (in inglese si legge "pipe").

Quindi forziamo l'assegnazione con un cast a `byte`:

```
jshell> b = (byte)128
b ==> -128
```

Come spiegato nel capitolo 3, il valore `128` immagazzinato nella variabile `byte b`, diventa `-128`. A causa del troncamento dei primi 24 bit della rappresentazione binaria di `128`. Proviamo quindi a sottrarre un'unità al numero che doveva essere `128`, e che invece risulta essere rappresentato in un `byte` come `-128`, e assegniamolo nuovamente alla variabile `b`. Anche in questo caso bisogna ricorrere al cast visto che un'espressione tra un tipo `byte` e un tipo `int` (la sottrazione `b-uno`) restituisce un risultato di tipo intero:

```
jshell> b = (byte)(b-uno)
b ==> 127
```

Il valore è ora correttamente immagazzinato nella variabile `byte`. Adesso eseguiamo il comando di stampa sommando `b` e di nuovo la variabile `uno`:

```
jshell> System.out.print(b+uno)
128
```

Questa volta viene stampato il valore corretto, visto che `b+uno`, è un'espressione tra un `byte` ed un `int`, e quindi il risultato viene promosso ad `int`, e `128` è un valore tranquillamente rappresentabile come tipo `int`.

Sino ad ora avevamo sempre utilizzato il metodo `println()` sull'oggetto `System.out`, invece questa volta abbiamo utilizzato il metodo `print()`. La differenza risiede nel fatto che `println()` stampa il valore in input e poi va a capo (`println` è l'abbreviativo di "print line"). Nel nostro esempio era inutile andare a capo e quindi ne abbiamo approfittato per introdurre il metodo `print()`.

Possiamo anche definire classi, enumerazioni, annotazioni ed anche metodi isolati, che probabilmente non sarà possibile scrivere su di un'unica riga come gli snippet scritti sinora. Ogni volta che andiamo a capo JShell, a meno di errori di sintassi, non valuterà quanto abbiamo scritto, ma aspetterà il completamento della struttura dati che stiamo definendo. Dobbiamo tener conto che per i tipi (classi, interfacce, enumerazioni ed annotazioni) non innestati saranno ignorati tutti i modificatori (tranne `abstract`), e che non è possibile dichiarare package o moduli. Infine, se chiamiamo un metodo che non abbiamo ancora definito, JShell ci avvertirà con un warning, ma ci consentirà di andare avanti considerando valido il nostro codice.

5.1.2.2 Regole JShell

Scrivere codice in JShell, presenta quindi alcune differenze rispetto a scriverlo all'interno di classi Java. Queste differenze hanno lo scopo di semplificare l'esperienza di codifica, ma contemporaneamente pongono anche dei limiti alla programmazione. Riassumiamo di seguito le principali differenze.

1. Benché sia possibile dichiarare metodi, non è richiesto obbligatoriamente dichiararne uno per scrivere uno statement.
2. Benché sia possibile dichiarare classi, non è richiesto obbligatoriamente dichiararne una per scrivere metodi o variabili.
3. Se scriviamo statement in un'unica linea, non è necessario utilizzare il simbolo `;`.
4. Non è possibile dichiarare package (cfr. capitolo 6) o moduli (cfr. capitolo 19).
5. Per tutti i tipi non innestati (cfr. modulo 10), tutti i modificatori fondamentali, a parte `abstract` (cfr. capitolo 7), sono ignorati o non permessi da JShell.
6. La parola chiave `synchronized` (cfr. capitolo 15) è sempre ignorata.
7. Dichiarare due volte la stessa variabile (o lo stesso metodo) causa la sovrascrittura della prima variabile dichiarata (o del primo metodo

dichiarato); nella programmazione standard questa situazione provocherebbe un errore in compilazione.

8. Dichiarare in uno snippet una variabile senza inizializzarla, implica che essa sia automaticamente inizializzata al proprio valore nullo.

5.1.3 Comandi JShell

I comandi JShell si distinguono dal codice Java perché sono sempre preceduti dal prefisso /. Sino ad ora abbiamo eseguito solo due comandi: il comando /help (equivalente a /?) ed il comando /exit. Cerchiamo di fare una breve panoramica dei comandi principali di Jshell.

5.1.3.1 Comandi esplorativi

Proviamo ad eseguire il comando **/history:**

```
jshell> /history

int uno = 1
boolean b
byte b
b = 128
b = (byte)128
b = (byte)(b-uno)
System.out.print(b+uno)
/history
```

Questo comando ci mostra la storia di tutto quello che abbiamo digitato in questa sessione (sia snippet che comandi Jshell). Se invece siamo interessati solo agli snippet possiamo utilizzare il comando **/list:**

```
jshell> /list

   1 :  int uno = 1;
   3 :  byte b;
   4 :  b = (byte)128
   5 :  b = (byte)(b-uno)
   6 :  System.out.println(b+uno)
```

Viene mostrato un elenco numerato degli snippet validi che abbiamo eseguito. Ogni snippet ha il suo **snippet id** (numero identificatore). Come si

può notare manca lo snippet id 2, dove avevamo dichiarato `b` come variabile booleana, ma che poi abbiamo sovrascritto con la definizione di `b` come variabile `byte` (snippet id 3). Non è stato riportato neanche lo snippet che ha provocato un errore. Notiamo inoltre come solo per gli snippet id 1 e 3 sia stato aggiunto il simbolo `;`. Questo perché JShell evidenzia il `;` solo quando si tratta di una dichiarazione. Gli snippet id possono essere richiamati con la sintassi:

```
/snippetId
```

Se richiamassimo uno snippet id relativo ad una dichiarazione di una variabile, essa verrà ridichiarata sovrascrivendo il valore che aveva magari assunto in un secondo momento. Il comando **/help** eseguito senza aggiungere opzioni elenca tutti i possibili comandi che possiamo eseguire con una breve spiegazione. È possibile invece avere qualche dettaglio in più su come funzionano i vari comandi facendo seguire al comando /help il nome del comando JShell di cui si desiderano le informazioni. In generale quindi possiamo digitare:

```
/help nomeComando
```

Per esempio:

```
jshell> /help /list
|
|  /list
|
|  Show the source of snippets, prefaced with the snippet
     id.
|
|  /list
|       List the currently active snippets of code that you
          typed or read with /open
|
|  /list -start
|       List the automatically evaluated start-up snippets
|
|  /list -all
|       List all snippets in cluding failed, overwritten,
          dropped, and start-up
|
```

```
|  /list <name>
|       List snippets with the specified name (preference for
         active snippets)
|
|  /list <id>
|       List the snippet with the specified snippet id
```

Scopriamo così che è possibile passare delle opzioni al comando /list. In particolare per vedere tutti i dettagli della nostra sessione, compresi gli snippet non validi e le istruzioni che ha implicitamente aggiunto JShell, possiamo eseguire il comando /list -all:

```
jshell> /list -all

..s1 :  import java.io.*;
  s2 :  import java.math.*;
  s3 :  import java.net.*;
  s4 :  import java.nio.file.*;
  s5 :  import java.util.*;
  s6 :  import java.util.concurrent.*;
  s7 :  import java.util.function.*;
  s8 :  import java.util.prefs.*;

  s9 :  import java.util.regex.*;
 s10 :  import java.util.stream.*;
   1 :  int uno = 1;
   2 :  boolean b;
   3 :  byte b;
  e1 :  b = 128
   4 :  b = (byte)128
   5 :  b = (byte)(b-uno)
   6 :  System.out.println(b+uno)
```

Da questo output possiamo capire varie cose. Per esempio che JShell importa varie librerie automaticamente per semplificarci la vita. Infatti esiste uno **script di startup** che importa automaticamente le librerie listate nell'esempio precedente, e che viene eseguito quando parte la sessione di JShell. Le librerie sono identificate con un id composto dalla lettera "s" (iniziale di "start") e un numero sequenziale, ovvero s1, s2, s3, etc. Vedremo nel paragrafo successivo come sia possibile cambiare queste impostazioni. Se vogliamo vedere solo i package importati possiamo usare il comando:

```
jshell> /list -start
```

Oppure il comando **/imports:**

```
jshell> /imports
```

che non ci mostrerà la numerazione s1, s2, s3, etc. accanto ad ogni istruzione di import. La differenza tra i due comandi risiede soprattutto nel fatto che quest'ultimo riporterà anche altri eventuali istruzioni di import digitate come snippet, mentre il primo riporterà solo gli import inclusi di default.

Possiamo anche notare che in questo output è presente anche lo snippet con id 2, anche se è stato sovrascritto dallo snippet con id 3. Inoltre viene listata anche l'istruzione (`b = 128`) che ha causato l'errore, a cui stato assegnato lo snippet id `e1` (la lettera "e" sta per "error"). I comandi **/types, /vars** e **/methods**, hanno la stessa funzione di /list, ovvero quella di elencare quanto già digitato. La differenza è che il comando /types mostra solo i tipi dichiarati (classi interfacce, annotazioni ed enumerazioni), il comando /methods mostra solo i metodi dichiarati (non quelli dichiarati nelle classi), il comando /vars mostra solo le variabili dichiarate (non quelle dichiarate nelle classi e nei metodi). Questi comandi condividono le stesse opzioni del comando /list: per esempio -all per vedere tutte le occorrenze valide e non valide (di tipi, variabili, metodi), -start per vedere cosa appartiene allo script di partenza della sessione (di tipi, variabili, metodi), oppure è possibile specificare lo snippet id o il nome (di tipi, variabili, metodi).

5.1.3.2 Comandi esecutivi

Abbiamo già asserito che esiste uno **script di startup** che importa automaticamente alcuni package, e che viene eseguito quando parte la sessione di JShell. In particolare abbiamo visto che di default JShell esegue uno script che importa i più comuni package (vedi esempio precedente).

Tale lista di `import` può essere configurata mediante il comando **/set start;** sono già disponibili tre configurazioni diverse, ma è anche possibile crearne di nuove. Oltre alla configurazione di default (che si chiama appunto `DEFAULT`), esistono altre due configurazioni predefinite: `JAVASE` e `PRINTING`. Con il comando:

```
jshell> /set start -retain PRINTING
```

otterremo che sarà impostato come script di startup per le prossime sessioni lo script PRINTING.

> **Se non specifichiamo l'opzione -retain, lo script di startup non sarà mantenuto per le prossime sessioni.**

Se vogliamo applicare lo script impostato nella sessione corrente da subito, dobbiamo eseguire il comando **/reload**, che rilancia anche tutte le istruzioni valide eseguite nella sessione. Lo script PRINTING metterà a disposizione dei metodi con nomi brevi per la stampa, e potremo usare in luogo di `System.out.println()`, `System.out.print()` e `System.out.printf()` (che vedremo più avanti), rispettivamente i metodi `println()`, `print()` e `printf()`. Il terzo script JAVASE invece importa tutti i package possibili di Java Standard Edition. In questo modo non saremo obbligati a importare mai nulla, ma dovremo aspettare qualche secondo in più prima che parta JShell. È possibile anche impostare più di uno script di startup alla volta, per esempio:

```
jshell> /set start -retain DEFAULT PRINTING
```

Per impostare script personalizzati invece, possiamo scrivere:

```
jshell> /set start -retain C:/MiaCartella/MioScript
```

Oppure andare a scrivere in un file l'attuale configurazione di startup con il comando **/save:**

```
jshell> /save -start C:/MiaCartella/MioScript
```

Con il comando /save possiamo anche salvare in un file altre informazioni. Per esempio possiamo salvare tutta la lista delle istruzioni valide che abbiamo digitato con:

```
jshell> /save C:/MiaCartella/MioScript
```

oppure tutta la lista delle istruzioni valide e non valide:

```
jshell> /save -all C:/MiaCartella/MioScript
```

oppure tutta la history della sessione corrente:

```
jshell> /save -history C:/MiaCartella/MioScript
```

Con il comando **/open** invece, possiamo aprire un file precedentemente salvato. Si può trattare anche di un file .java. Per esempio:

```
jshell> /open C:/MiaCartella/HelloWorld.java
```

Con **/env** possiamo impostare la variabile CLASSPATH (solo introdotta nel primo capitolo e dettagliata nell'appendice E) per farla puntare a qualche libreria (file .jar), con la sintassi:

```
jshell> /env –class-path C:/MiaCartella/MiaLibreria.jar
```

Con /env è possibile anche impostare il cosiddetto MODULEPATH, che però sarà argomento dell'ultimo modulo di questo libro.

Il comando **/drop** può eliminare uno snippet, specificando il suo nome o il suo id. Con il comando **/reset** invece, il comando /list sarà ripulito di tutte le istruzioni eseguite. Il comando /history invece continuerà a funzionare come funzionava precedentemente l'esecuzione del comando /reset.

È possibile richiamare l'ultima riga (valida o non valida) editata anche con il comando /!.

5.1.4 Strumenti ausiliari di JShell

Siccome JShell viene eseguito all'interno di una prompt DOS, è possibile sfruttarne alcune comodità. Per esempio possiamo **richiamare** precedenti righe digitate anche usando il tasto di Freccia Su ↑ (e tornare indietro con il tasto Freccia Giù ↓). È possibile anche **copiare, incollare** e **ricercare** mediante le funzionalità della prompt DOS, come descritto nell'appendice C.

Ma questi non sono gli unici aiuti di cui si può usufruire utilizzando JShell.

5.1.4.1 Variabili implicite

JShell assegna un reference o una variabile primitiva implicitamente, nel caso non ne assegniamo noi una esplicitamente, deducendone il tipo automaticamente. Si parla di **variabili implicite con tipo dedotto (implicit**

variables with type inferred). Per esempio se scriviamo:

```
jshell> 1
$1 ==> 1
```

possiamo notare che a `1` viene assegnata implicitamente la variabile intera `$1` (il tipo dedotto automaticamente). Possiamo richiamare la variabile ed utilizzarla, per esempio:

```
jshell> $1 + $1
$2 ==> 2
```

si noti come sia stata creata anche una seconda variabile implicita `($2)`.

5.1.4.2 Forwarding reference

Abbiamo già asserito che possiamo dichiarare anche tipi o metodi, e che magari possiamo anche copiarli in JShell a partire da codice già scritto. In casi come questi potrebbe capitare di invocare metodi, o utilizzare classi, non ancora definiti. Per esempio, se dichiarassimo un reference di una classe non ancora dichiarata:

```
jshell> Utente u;
|  created variable u, however, it cannot be referenced
     until class Utente is declared
```

JShell la creerebbe, ma ci avvertirebbe che non sarà possibile utilizzarla fin quando la classe `Utente` non sarà dichiarata. Se creassimo la classe `Utente` successivamente:

```
jshell> class Utente {}
|  created class Utente
|    update replaced variable u, reset to null
```

la variabile `u` sarebbe inizializzata al suo valore di default `(null)`. Un discorso simile si ripete se proviamo a chiamare un metodo non ancora definito:

```
jshell> void testForward() {
   ...> stampa();
   ...> }
|  created method testForward(), however, it cannot be
```

```
      invoked until method stampa() is declared
```

5.1.4.3 Auto-completamento

Il tasto TAB, può essere utilizzato per l'**auto-completamento**, sia per snippet che per comandi JShell. Per esempio, se dopo aver scritto **Strin** battiamo il tasto TAB, otterremo il seguente output:

```
jshell> String
String                              StringBuffer
StringBufferInputStream             StringBuilder
StringIndexOutOfBoundsException     StringJoiner
StringReader                        StringTokenizer
StringWriter
```

Quindi ci vengono proposte tutte le possibili opzioni per completare il nostro codice, ma intanto viene scelto quello più probabile. TAB può essere usato ogni volta che si vuole un aiuto o che si vuole esplorare una particolare classe, libreria o metodo, e anche un comando JShell. Con una doppia pressione del tasto TAB si ottiene la **documentazione ufficiale** dell'ultima dichiarazione, da scorrere pagina dopo pagina sempre con il tasto TAB.

Un altro ausilio per completare automaticamente il nostro codice, JShell ce lo offre con un'insolita combinazione di tasti. Per **importare automaticamente** una classe non ancora importata per esempio, bisogna prima utilizzare tale classe. Poi premere contemporaneamente i tasti **SHIFT** e **TAB**, rilasciarli e dopo premere il tasto i (che sta per "import"). Per esempio, se utilizziamo questa combinazione di tasti dopo aver digitato:

```
jshell> new Scene
```

allora JShell ci proporrà di scegliere tra gli import possibili o di non fare niente (opzione **0**):

```
0: Do nothing
1: import: javafx.scene.Scene
Choice:
Imported: javafx.scene.Scene
```

Oppure, se dopo aver dichiarato un valore, vogliamo **dichiarare un reference** (o una variabile primitiva) basterà semplicemente premere

contemporaneamente i tasti **SHIFT** e **TAB**, rilasciarli, quindi premere il tasto **v** (che sta per “variable”).JShell dedurrà il tipo della variabile, la dichiarerà e posizionerà il cursore subito dopo per permetterci di definire il reference. Infatti dopo aver istanziato un `Integer`, utilizzando la suddetta combinazione di tasti otteniamo:

```
jshell> new Integer(1);
jshell> Integer = new Integer(1);
```

dove il cursore è posizionato prima del simbolo =. Non ci resta che scrivere il nome del reference e confermare:

```
jshell> Integer intero = new Integer(1);
intero ==> 1
```

5.1.4.4 Il comando /edit

Il comando **/edit** apre un semplice editor chiamato **JShell Edit Pad** (vedi figura 5.2), che in alcuni casi (come nei copia-incolla) semplifica lo sviluppo del codice.

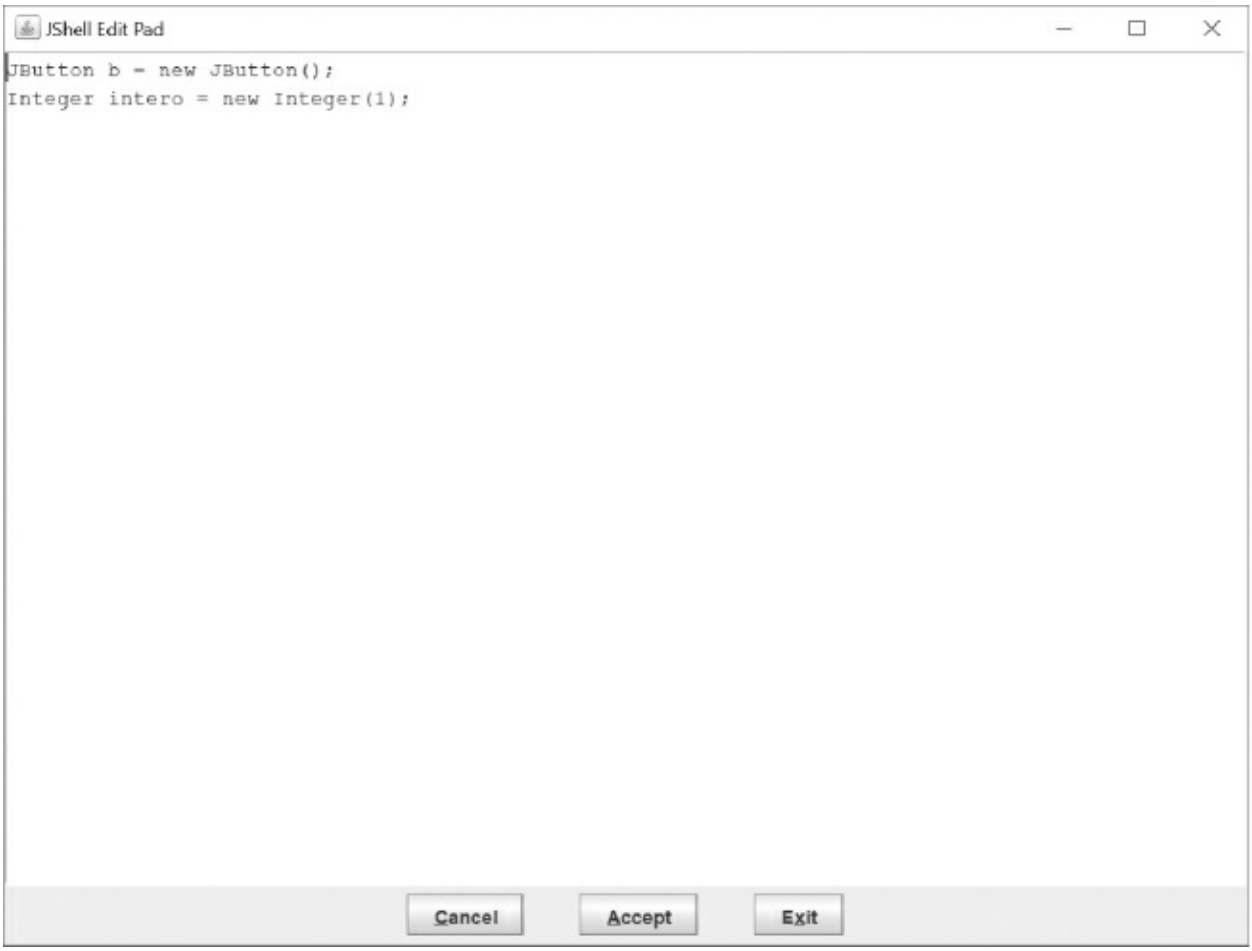


Figura 5.2 - JShell Edit Pad in azione.

Tutto quello che scriveremo in JShell Edit Pad può essere riportato al volo su JShell facendo clic sul bottone **Accept** (JShell si aggiornerà contestualmente con i nuovi valori editati) oppure sarà riportato automaticamente dopo che si è chiuso JShell Edit Pad facendo clic sul bottone **Exit**.

> **Cancellare istruzioni già presenti non significa eliminarle come se si stesse usando il comando /drop. Semplicemente non le vedremo nell'editor.**

Le modifiche saranno ignorate se facciamo clic su **Cancel.**

Se vogliamo sfruttare un altro editor più completo (per esempio Notepad++) è possibile impostarlo come default per la sessione corrente (per l'impostazione permanente utilizzare l'opzione -retain) con il comando /set:

```
jshell> /set editor C:\Program Files (x86)\Notepad++\
  notepad++.exe
|  Editor set to: C:\Program Files (x86)\Notepad++\
     notepad++.exe
```

Salvando il file in Notepad++, otterremo lo stesso effetto del bottone Accept di JShell Edit Pad. Con il comando **/set** è possibile anche configurare altre funzionalità. Per esempio con l'opzione **feedback**, è possibile aumentare la verbosità dei messaggi di analisi specificando il valore **verbose** (saranno descritte testualmente le operazioni eseguite), oppure diminuirla specificando il valore **coincise** (saranno stampati solo i risultati delle operazioni e non delle dichiarazioni), o evitarli del tutto con il comando **silent**. Per i più esigenti esistono anche le opzioni, **mode, format, prompt** e **truncation**, per personalizzare ulteriormente JShell. Si rimanda alla documentazione (comando **/help**) se si è interessati a questi dettagli.

5.1.4.5 Scorciatoie da tastiera (shortcuts)

JShell offre alcune **scorciatoie da tastiera**, in inglese **shortcuts** (ovvero combinazioni di tasti) per automatizzare dei **comandi predefiniti.** Segue una tabella esplicativa dei comandi disponibili (i tasti vanno premuti contemporaneamente):

Shortcut	Descrizione
CTRL a	Sposta il cursore all'inizio della riga
CTRL e	Sposta il cursore alla fine della riga
ALT b	Sposta il cursore un carattere indietro
ALT f	Sposta il cursore un carattere avanti
CTRL b	Sposta il cursore all'inizio della parola precedente
CTRL f	Sposta il cursore all'inizio della parola successiva
CTRL r	Esegue una ricerca dell'ultimo comando o snippet che contiene i caratteri digitati subito dopo CTRL r
CTRL t	Scambia i due caratteri alla sinistra del cursore

CTRL w	Cancella la parola alla sinistra del cursore
ALT d	Cancella la parola alla destra del cursore
CTRL k	Cancella tutto quanto è alla destra del cursore fino alla fine della riga
CTRL u	Cancella tutto quanto è alla sinistra del cursore fino alla fine della riga

5.2 Integrated Development Environment (IDE)

Un **Integrated Development Environment** (in italiano **ambiente di sviluppo integrato)**, detto più brevemente **IDE**, è un software che integra vari strumenti per sviluppare. Per quanto riguarda Java, gli IDE più utilizzati sono sicuramente **Eclipse** e **Netbeans.** Essi mettono a disposizione un compilatore, un editor avanzato, strumenti di completamento codice, una JVM, un debugger, un generatore di archivi, e così via, il tutto all'interno di uno strumento relativamente semplice da utilizzare con un'interfaccia grafica intuitiva. Quest'ultima è costituita da tantissime aree diverse, tab, menu, menu a tendina, pulsanti e finestre di dialogo che possono essere visualizzate o meno all'occorrenza.

Entrambi i software sono **free** ed **open source**, dove con free intendiamo gratuiti, e con open source intendiamo che il codice sorgente è liberamente scaricabile. Entrambi supportano Java 8, mentre attualmente (gennaio 2018) il supporto a Java 9 non è garantito. Per Eclipse esiste un plugin provvisorio per Java 9 (https://marketplace.eclipse.org/content/java-9-support-beta-oxygen), mentre per Netbeans è possibile scaricare una versione non stabile (http://wiki.netbeans.org/JDK9Support).

Eclipse è sviluppato dalla Eclipse Foundation ed è scaricabile all'indirizzo http://www.eclipse.org. A partire dal suo nucleo sono stati creati tanti altri IDE a pagamento professionali da varie aziende, che mettono a disposizione strumenti ancora più evoluti per sviluppare e integrazioni con altri software proprietari. Permette personalizzazioni estreme, continui aggiornamenti, integrazioni con software di tutti i tipi mediante plug-in, ed una stabilità di base eccellente.

Netbeans è stato lo strumento di sviluppo ufficiale consigliato da Oracle

fino al 2016, quando il suo progetto è stato donato alla Apache Software Foundation. È scaricabile all'indirizzo https://netbeans.org. Questo momento di transizione ha provocato la stasi delle revisioni dell'IDE che da mesi non viene più aggiornato. Il punto di forza di Netbeans è la semplicità. Rispetto ad Eclipse sembra avere qualche problema in più, ma la curva di apprendimento è decisamente meno ripida rispetto a quella di Eclipse.

Un altro IDE molto utilizzato è **IntelliJ IDEA** della JetBrains (scaricabile all'indirizzo https://www.jetbrains.com/idea). Quest'ultimo però rende disponibile per il download gratuito solo la versione Community (poi esiste una versione di valutazione che dopo un periodo di tempo bisogna acquistare). Per gli scopi di questo libro la versione Community va benissimo, ed al momento è l'unico IDE che supporta lo sviluppo di Java 9 ufficialmente. La scelta dell'IDE è personale ma è comunque altamente probabile che nella carriera da programmatore non si abbia a che fare con un unico IDE.

5.2.1 Progetto

Quando utilizziamo un IDE non possiamo creare codice al volo così facilmente come abbiamo visto con JShell. Il concetto di unità di sviluppo che accomuna gli IDE, è il concetto di **progetto.** Si deve creare prima un progetto, che risiede in una determinata cartella con un certo nome sotto cui si sviluppa automaticamente (secondo la filosofia dell'IDE) l'alberatura delle sottocartelle che conterranno, separati tra loro, i sorgenti, i file .class, le risorse, le librerie, i file di configurazione e così via. Da un certo punto di vista questo è un grande vantaggio, perché l'IDE si occupa in maniera semiautomatica dei file, liberandoci da una gestione che potrebbe diventare un inferno. La **compilazione** è spesso fatta **in line**, ovvero mentre si scrive il codice, l'IDE segnalerà eventuali errori o warning al volo, senza aspettare una eventuale compilazione esplicita mediante la pressione di un bottone (in Eclipse non esiste neanche il bottone per compilare).

5.2.2 Editor

L'**editor**, ovvero l'area di testo dove è possibile scrivere il codice, supporta decine di tecniche che consentono una codifica più veloce e corretta. Esistono

tantissime combinazioni di tasti che permettono di creare anche frammenti di codice piuttosto complessi. La più famosa è la combinazione dei tasti CTRL e BARRA SPAZIATRICE, che provoca l'auto-completamento del codice che si è iniziato a scrivere, o nel caso di più opzioni disponibili, la scelta tra esse (simile al tasto TAB per JShell). Inoltre la pressione del tasto destro del mouse sull'editor (su ogni IDE) fa visualizzare un menu a tendina con varie opzioni. Tra queste si trovano anche diverse che eseguono tecniche di **refactoring del codice**, ovvero tecniche standard che modificano l'aspetto del codice senza modificarne la funzionalità. È un po' come applicare dei pattern di programmazione migliorando la leggibilità e la qualità del codice.

5.2.3 Debugger

Uno degli strumenti più utili di un IDE è indubbiamente il **debugger.** Si tratta di uno strumento che ci permette di eseguire l'applicazione in una modalità speciale, che consente l'introspezione dei valori delle variabili al runtime. Infatti è possibile inserire, sul lato sinistro del nostro editor (in corrispondenza dei numeri di riga), i cosiddetti **breakpoint.** Quando il flusso dell'applicazione giunge ad una riga marcata con un breakpoint, il programma si blocca, come se si fermasse il tempo. A quel punto l'IDE ci offrirà varie possibilità: dall'ispezione dei valori delle variabili e degli oggetti disponibili in quel punto di codice, alla possibilità di valutare espressioni che non abbiamo scritto (per esempio chiamare un metodo differente rispetto a quello definito nel codice). Dalla possibilità di esplorare la coda di chiamate tra i vari oggetti (per capire quale oggetto ha chiamato il metodo in cui ci siamo fermati col debugger), alla possibilità di cambiare i valori delle variabili per continuare l'esecuzione del programma in maniera diversa. Poi sarà possibile muoverci con il debugger: possiamo andare alla riga successiva, al prossimo punto di debug, entrare in un metodo chiamato nella riga corrente, o uscire dal metodo dove ci si trova per tornare al metodo chiamante, e così via. In programmi complessi e applicazioni professionali un debugger è uno strumento indispensabile.

5.2.4 Integrazioni

Un IDE permette di integrarsi con altri strumenti di sviluppo ed utilizzarli

direttamente dalla sua interfaccia grafica. Infatti un programmatore Java non scrive solo codice, soprattutto se lavora in azienda insieme ad altri sviluppatori. La maggior parte delle applicazioni per esempio interagisce con un **database** (in italiano **base di dati)** in cui memorizza e recupera i dati da conservare. Qualsiasi sia il database (Oracle, MySQL, Postgres e così via) un IDE fornirà un modo per connettersi, consultarlo e aggiornarlo da una delle sue interfacce. Eclipse per esempio, può cambiare completamente il layout di visualizzazione con bottoni dedicati chiamati debug, database, server, etc. Un'altra tipica integrazione che offre un IDE, è con un **application server**, ovvero un software col quale possono essere eseguiti particolari tipi di applicazioni. Un esempio potrebbe essere il server Oracle Weblogic, che è un application server Java EE (Enterprise Edition). Questo è capace di far girare applicazioni della tecnologia Java EE, come servlet, EJB, JSP e così via, gestendone il ciclo di vita. Tali applicazioni infatti non definiscono un metodo `main()`, bensì hanno delle caratteristiche tali da poter essere gestite da un application server. Un IDE può avviare, spegnere e configurare un application server dalla sua interfaccia grafica. Tra le tante altre integrazioni che si possono trovare all'interno di un IDE, ricordiamo i **version control systems** (per esempio GIT, Mercurial, Subversion, etc.), che gestiscono le versioni (e i problemi legati ad esse) del codice condiviso da più sviluppatori.

5.3 Realtà aziendale

Per diventare uno sviluppatore in un'azienda informatica purtroppo non basta sapere programmare in Java. Quando si entra in azienda, anche se magari si possiede una laurea in informatica o in ingegneria informatica, è facile trovare diverse difficoltà. Meglio quindi avere al più presto un quadro della situazione, per capire come muoversi al meglio.

5.3.1 Architettura

Quando si entra in azienda, un aspetto a volte addirittura sconosciuto del software è quello dell'architettura. Quando si programma solitamente ci si concentra sulla parte funzionale dell'applicazione. Individuiamo le funzionalità dell'applicazione, cerchiamo di capire il modo migliore per risolvere i problemi, ed implementiamo una soluzione. L'architettura

racchiude tutte le attività che risolvono i problemi dei **requisiti non funzionali.** Cerchiamo di fare chiarezza con un esempio usando UML, in particolare un diagramma dei componenti innestato in un diagramma di deployment. Il **diagramma dei componenti** (in inglese **component diagram)** è un diagramma statico che descrive i componenti software e le loro dipendenze. Il **componente software** (in inglese chiamato semplicemente **component)** è la notazione protagonista di questo diagramma, e può rappresentare un qualsiasi tipo di programma (o parte di programma, libreria, tecnologia, etc.). La sua notazione grafica è costituita da un rettangolo con due piccoli rettangoli che fuoriescono dall'angolo superiore sinistro. Il **deployment diagram** (che in italiano potremmo tradurre come **diagramma di installazione)** mostra i componenti hardware dove il software sarà installato. In questo diagramma l'elemento principale si chiama **nodo**, è raffigurato come un cubo, e rappresenta un **componente hardware.** Le frecce tratteggiate implicano **dipendenza** del componente da cui parte la freccia al componente indicato dalla freccia. La notazione di dipendenza in UML è generica e può essere usata in tutti i diagrammi, ma in questo caso intendiamo che il componente da cui parte la freccia, ha bisogno del componente indicato dalla freccia per funzionare.

Con il diagramma di figura 5.3, mostriamo essenzialmente dove vengono installati ed eseguiti i componenti software che compongono un'applicazione gestionale, ovvero un'applicazione che supporta la gestione di un'attività con un magazzino (come per esempio un negozio). Si noti come con questo diagramma si riesce a dare una visione di alto livello dell'architettura di questa applicazione. Si possono capire varie cose. In primis che l'applicazione è composta da due parti ben distinte, una parte client e una parte server. Per definizione un **client** è un programma che richiede servizi ad un altro programma che si chiama server. Per definizione un **server**, è un programma sempre in esecuzione, che mette a disposizione dei servizi. Un client e un server comunicano solitamente tramite rete, con un protocollo ben definito (nell'esempio è stato specificato il protocollo REST).

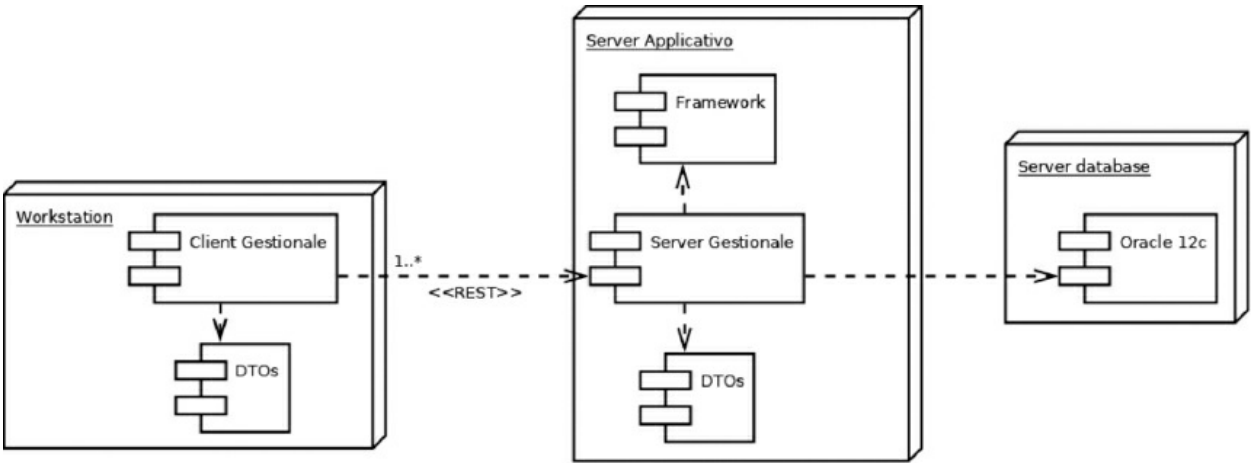


Figura 5.3 - Deployment Diagram di un gestionale.

> **Con il termine "server" spesso ci si riferisce anche alla macchina dove gira l'applicazione server, perché solitamente questa macchina è dedicata ad ospitare solo quel programma.**

La notazione di molteplicità 1..* (cfr. appendice L) indica che esistono un server e uno o più client. Inoltre condividono un componente chiamato DTOs, che sarà installato su entrambi i computer e da cui dipendono entrambi i componenti client e server. Si noti come il componente server acceda ad un database, mentre i client non possono accedervi, ma possono richiedere al server i vari servizi che mette a disposizione tramite il protocollo REST. Inoltre il server dipende da un **framework** (non abbiamo specificato un nome). Un framework è uno strato software su cui si può basare un'applicazione, seguendone le funzionalità e le caratteristiche, per non cominciare a lavorare da zero. I framework spesso semplificano lo sviluppo, ma questo purtroppo non accade sempre! L'acronimo **DTO** in particolare, sta per **Data Transfer Object** che è il nome di un famoso pattern architetturale utilizzato per minimizzare le interazioni di rete client-server, creando oggetti per trasportare dati che aggregano informazioni da oggetti diversi. Quindi con il nome DTOs (la "s" finale sta ad indicare il plurale di DTO) aggiungiamo un'ulteriore informazione al diagramma. L'architettura ha, tra i suoi scopi, il soddisfare il requisito non funzionale dell'ottimizzazione delle prestazioni.

Un programmatore del sistema rappresentato in figura 5.3, deve comprendere appieno l'architettura su cui sta lavorando. Per esempio sviluppando una classe all'interno del componente DTOs, deve essere consapevole che essa sarà utilizzata sia dal programma client che dal programma server. Può sembrare scontato, ma in realtà non lo è affatto.

5.3.2 Conoscenze

Java è un linguaggio di grande successo anche perché si adatta e riesce ad integrarsi con altri linguaggi e tecnologie. Quindi in primo luogo, è quasi impossibile non avere **altre nozioni informatiche** importanti. Per esempio conoscere un minimo di teoria sui database relazionali, e le basi del linguaggio SQL per interrogarli è praticamente obbligatorio. Di conseguenza è molto probabile anche che siano richieste conoscenze di framework che sono considerati standard per l'interazione con database come EJB o Hibernate (standard JPA). Anche conoscere la teoria di base del linguaggio XML è fondamentale (a tal riguardo è possibile consultare l'appendice H, e poi l'appendice N per il supporto di Java all'XML). Se si lavora nel campo web, avere delle conoscenze basiche sul protocollo HTTP, i linguaggi HTML, Javascript e CSS, librerie come Bootstrap, e framework come Spring, o Angular JS è quasi scontato.

Un altro aspetto da non sottovalutare, è quello della **conoscenza del business.** Quando si crea un programma per un cliente, solitamente bisogna imparare il business in cui sarà usato il software in breve tempo, come funziona il business del cliente, e trovare delle soluzioni che possano semplificare il suo lavoro. Si impara una terminologia sconosciuta e si acquisiscono conoscenze sul flusso di lavoro del cliente, durante l'intera durata del progetto.

Infine è da sottolineare che sviluppare insieme ad **altre persone** può risultare complicato, perché le divergenze esistono sempre, a volte manca empatia, e spesso si lavora su codice scritto da altri che magari può risultare poco intuitivo, poco leggibile, fuori dagli standard conosciuti, o in alcuni casi da rifare!

5.3.3 Tipologie di programmi

Sino ad ora abbiamo creato essenzialmente programmi brevi che servono per imparare i vari concetti. Questo tipo di applicazioni che abbiamo sviluppato ricadono tutte sotto una certa categoria: le **applicazioni standalone** (dette anche **applicazioni desktop).** Questo tipo di applicazioni sono eseguite su computer desktop e portatili e solitamente hanno un'interfaccia grafica. Come abbiamo imparato, esse iniziano eseguendo un metodo `main()`, che come sappiamo va eseguito da riga di comando tramite la JVM. Quando però distribuiamo un software non possiamo pretendere che l'utente per eseguire un'applicazione Java debba aprire una prompt dei comandi. Piuttosto bisogna fornire un file capace di eseguire il nostro codice, con un semplice doppio click. Il modo più semplice è rilasciare uno script di esecuzione all'interno di un file batch (file con suffisso .bat). Un esempio di programma standalone è EJE, che viene eseguito mediante il file batch denominato EJE.bat. Questo file contiene il seguente script:

```
  start /MIN "EJE running..."
    javaw -splash:resources/images/splash.png
com.cdsc.eje.gui.EJE
```

che esegue EJE con il comando javaw (che è la versione specializzata del comando java per applicazioni con interfaccia grafica) specificando una splash screen (ovvero un'immagine che viene visualizzata durante il caricamento delle risorse in fase di avvio del programma) e nascondendo la prompt DOS utilizzata per eseguire il comando.

Oggigiorno però, le applicazioni standalone sono solo una piccola parte dell'insieme di applicazioni che si sviluppano professionalmente. Il termine "standalone" può tradursi come "autonoma". Infatti un'applicazione standalone, a parte l'installazione della JVM, non richiede altro per essere eseguita. Gli altri tipi di applicazioni invece girano all'interno di contesti di esecuzione. Le **applicazioni web** per esempio sono applicazioni che hanno un'architettura suddivisa in una parte client e una parte server. Un **client web** richiede pagine web e coincide con i programmi che chiamiamo comunemente **browser** (Mozilla Firefox, Google Chrome e così via). Un **server web**, è invece un'applicazione che mette a disposizione servizi disponibili in rete. Un famoso esempio di server web è Apache HTTP Server (l**'HTTP** è il principale protocollo di comunicazione della rete, e che è

utilizzato dai browser). All'interno del web server eseguono applicazioni che creano pagine web con varie tecnologie tra cui Java, Javascript, HTML e altre, dette appunto applicazioni web. Il ciclo di vita di un'applicazione web è gestito dal web server (ovvero non partono eseguendo un metodo `main()`) così come la sua configurazione.

Le **applicazioni enterprise** (che potremmo tradurre come applicazioni aziendali) sono un'evoluzione delle applicazioni web, e solitamente mettono a disposizione servizi più complessi come scaricamento di risorse, **web services** (ovvero applicazioni di comunicazione tra sistemi eterogenei che usano il protocollo HTTP), servizi di reportistica, etc., e che quindi come **client enterprise** possono avere anche programmi appositamente creati per interagire con lo strato **server enterprise.** Quest'ultimo è costituito a sua volte da vari strati che utilizzano tecnologie diverse per adempiere a vari scopi (l'applicazione gestionale raffigurata in figura 5.3 può essere considerata di tipo enterprise).

Ma il tipo di applicazione che probabilmente oggi è la più sviluppata è l**'applicazione mobile.** Negli ultimi anni con l'incredibile diffusione di tablet e smartphone, sono state sviluppate miliardi di cosiddette **app**, che hanno rivoluzionato il modo in cui sviluppiamo e concepiamo il software. Questo tipo di applicazione può avere sia un'architettura client-server, sia un'architettura solo client. La parte client è particolarmente importante, e se a volte è sviluppata con pagine web per garantire la massima portabilità tra i vari dispositivi mobili, altre volte viene sviluppata nel linguaggio nativo delle piattaforme a cui è destinato (come detto per Android si programma in Java).

5.3.4 Ruoli

Quando si sviluppa in una realtà aziendale, programmare è solo una parte delle attività che vengono svolte. L**'ingegneria del software** definisce diverse attività che concorrono alla realizzazione di un software funzionante. In azienda non ci sono solo sviluppatori, ma esistono tante persone che interpretano vari ruoli, ognuno dei quali dovrebbe avere responsabilità precise all'interno delle diverse fasi del ciclo di sviluppo del software. Il **capo progetto** (in inglese **project manager)** dovrebbe pianificare il lavoro a livello temporale, assegnare il lavoro alle risorse, interagire con tutti i principali membri del gruppo di sviluppo, con il management e con il cliente.

Il **business analyst** invece ha lo scopo di raccogliere requisiti dal cliente, formarlo e risolvere dubbi funzionali. Il ruolo dell'**architetto del software** consiste nel gestire le attività non funzionali, ovvero nello scegliere le tecnologie, nel supervisionare la qualità del codice, nel migliorare l'usabilità del software, nel decidere l'architettura da usare, nel decidere la struttura e l'installazione dell'applicazione, nel migliorare le prestazioni, nel formare gli sviluppatori e così via. L'**IT manager** (in italiano meglio noto come **sistemista)** si occupa di preparare gli ambienti, di installare i software, di configurare la rete, le macchine, i permessi, etc. Il **deployer** o **release manager**, ha la responsabilità di creare e installare le versioni del software per i vari ambienti in cui deve essere utilizzato. Il **tester** si occupa di scovare i bug nell'ambiente di collaudo e riportarli ai responsabili del gruppo di sviluppo allo scopo di risoluzione. Il **database administrator** (o **DBA)** è l'esperto del database e si occupa di tutto ciò che concerne l'interazione e la configurazione del database. Il **grafico** è il responsabile dell'aspetto della grafica dell'applicazione. Lo stesso **sviluppatore** avrà varie attività da eseguire, a seconda dell'organizzazione aziendale. Se l'azienda non è molto strutturata, capita spesso che debba saper fare analisi, progettazione e documentazione oltre allo sviluppo. Più piccola è l'azienda, e più è facile che più ruoli siano assegnati ad un'unica persona. Insomma dall'organizzazione aziendale, e dalle risorse disponibili dipende a chi saranno assegnate le varie attività.

5.3.5 Metodologia

Tra queste attività ci sono: la **raccolta dei requisiti** con il cliente, la formalizzazione in un documento come un **SRS (Software Requirement Specification**, ovvero **specifiche di requisiti software)**, l'**analisi degli scenari**, la **scelta delle tecnologie**, la scelta e la preparazione degli **ambienti** di sviluppo, collaudo e manutenzione, il piano di **fattibilità** rispetto alle risorse disponibili, il **piano di progetto** che include tutte le attività e le relative pianificazioni temporali, la progettazione della **soluzione architetturale**, la **progettazione** delle funzionalità, la **progettazione del database**, la creazione della **documentazione tecnica interna**, la creazione dell'ambiente di **condivisione delle informazioni**, lo **sviluppo**, l'**installazione**, il **collaudo** (i test), la **risoluzione dei bug**, la **formazione**

all'utente, la creazione del **manuale utente**, la **manutenzione** e l'**evoluzione** del software. Quindi come è possibile notare, imparare a programmare è solo una piccola parte del tutto!

Inoltre tutte queste attività devono essere organizzate ed eseguite con certi standard, in un particolare ordine, usando una tempistica adeguata e le giuste tecniche. Tutto questo è solitamente regolamentato da una **metodologia aziendale.** Ogni azienda infatti adotta una propria metodologia, la quale può essere più o meno ispirata da metodologie standard affermate. Una **metodologia object oriented**, nella sua definizione più generale, potrebbe intendersi come una coppia costituita da un processo e da un linguaggio di modellazione.

A sua volta un **processo** potrebbe essere definito come la serie di indicazioni riguardanti i passi da intraprendere per portare a termine con successo un progetto. Un **linguaggio di modellazione** è invece lo strumento che le metodologie utilizzano per descrivere (possibilmente in maniera grafica) tutte le caratteristiche statiche e dinamiche di un progetto. **UML** non è altro che un linguaggio di modellazione, ed è considerato lo standard de facto dalla maggior parte delle metodologie affermate. Come abbiamo visto esso è costituito per linee generali da una serie di diagrammi grafici i cui elementi sono semplici linee, triangoli, rettangoli, omini stilizzati e così via. Questi diagrammi hanno il compito di descrivere in modo chiaro, tutto ciò che durante un progetto potrebbe risultare difficile o troppo lungo con documentazione testuale.

Non essendo questo un libro sull'ingegneria del software, non possiamo dedicare troppo spazio alla teoria. Quindi tra gli esercizi di questo capitolo, abbiamo introdotto un caso di studio che proveremo a sviluppare, capitolo dopo capitolo. In questo modo avremo degli esempi pratici su come si possano affrontare queste attività. Siccome sino ad ora abbiamo parlato quasi esclusivamente di programmazione Java, a partire dal prossimo paragrafo introdurremo con un esempio pratico uno dei tanti modi per approcciare al problema. Infatti nonostante ci si possa sentire abbastanza padroni dei concetti studiati, sino ad ora non abbiamo ancora un metodo per utilizzarli nella maniera corretta. Ovvero, per quanto il lettore abbia diligentemente studiato, non è probabilmente ancora in grado di sviluppare un programma in modo corretto. Negli esercizi infatti, non è mai stata richiesta l'implementazione di un'applicazione seppur semplice da zero. Per esempio,

vi sentite in grado di creare un'applicazione che simuli una rubrica telefonica? Quali classi creereste? Quante classi creereste? Sembra molto difficile già iniziare! Quindi introduciamo con la pratica un possibile **processo** che ci porti a qualche risultato concreto.

5.4 Caso di studio

Mettiamoci nei panni di un programmatore dipendente di una piccola azienda informatica che chiameremo **Noumeno S.r.l..** A questa piccola azienda viene commissionata un'applicazione web da parte di una catena di negozi di abbigliamento che ha cinque sedi sparse in varie città italiane, e che chiameremo **Panta Rei Clothing.** Viene stipulato un contratto che accetta di mettere on line nel giro di sei mesi, il nuovo gestionale interno di Panta Rei che sarà denominato **Logos.** Un**'applicazione gestionale** è un tipo di applicazione con architettura client-server (solitamente), che supporta la gestione di un'attività con un magazzino. Quest'applicazione deve permettere di ricercare tutti i prodotti vendibili, in tutti i negozi della catena mostrandone la disponibilità. I dipendenti dei vari negozi saranno gli utenti dell'applicazione e potranno anche inserire nuovi prodotti, modificarne la disponibilità, e cancellarli.

Una descrizione come la precedente (che potremmo chiamare **problem statement)** è insufficiente per poter iniziare a scrivere del codice. Quali classi creare tanto per iniziare? Siete in grado di utilizzare precedenti esperienze (magari sfruttando framework come Spring) per iniziare a buttare giù una soluzione di massima? Dal nostro punto di vista... assolutamente no!

I requisiti sono troppo pochi e troppo generici per poter iniziare a scrivere del codice, a scegliere un framework, ma anche solo per poter definire le funzionalità di base che l'applicazione dovrà avere. Introduciamo quindi il primo passo del nostro processo.

5.4.1 Raccolta requisiti con Use case e scenari

Senza individuare le funzionalità dell'applicazione è impossibile creare un programma. Quindi il nostro primo scopo è quello di individuarle, parlando

> con il cliente, capendo come funziona attualmente il business e cosa si vuole migliorare con il sistema richiesto. Possiamo intanto

introdurre una tecnica chiamata **analisi dei casi d'uso (use case analisys** in inglese). Utilizzeremo a tal proposito il **diagramma dei casi d'uso** (in inglese **use case diagram)** UML. Tale diagramma serve per rappresentare le interazioni tra il sistema e gli utenti del sistema stesso. Gli utenti del sistema sono rappresentati come omini stilizzati, e vengono detti **attori** (in inglese **actors)**, ma tale termine va interpretato come *ruoli* e non come persone fisiche. I **casi d'uso**, che per semplicità possiamo identificare con le funzionalità dell'applicazione, sono rappresentati da semplici ovali. Inoltre su questo diagramma è possibile riportare alcune **dipendenze** (frecce tratteggiate) ed altre **relazioni complesse** tra use case come la relazione **include**, che implica che il caso d'uso da cui parte la freccia potrebbe utilizzare il caso d'uso puntato dalla freccia (cfr. appendice L). Laddove si vuole esprimere un concetto in maniera testuale si può utilizzare una **nota** (foglietto con angolo piegato), che è un elemento comune a tutti i diagrammi UML. Infine il cosiddetto **system boundary** (in italiano confine del sistema) che rappresenta il sistema stesso, e viene rappresentato con un rettangolo (contiene i casi d'uso).

5.4.1.1 Use case diagram

Nella figura 5.4 riportiamo il diagramma finale derivante dalla nostra analisi.

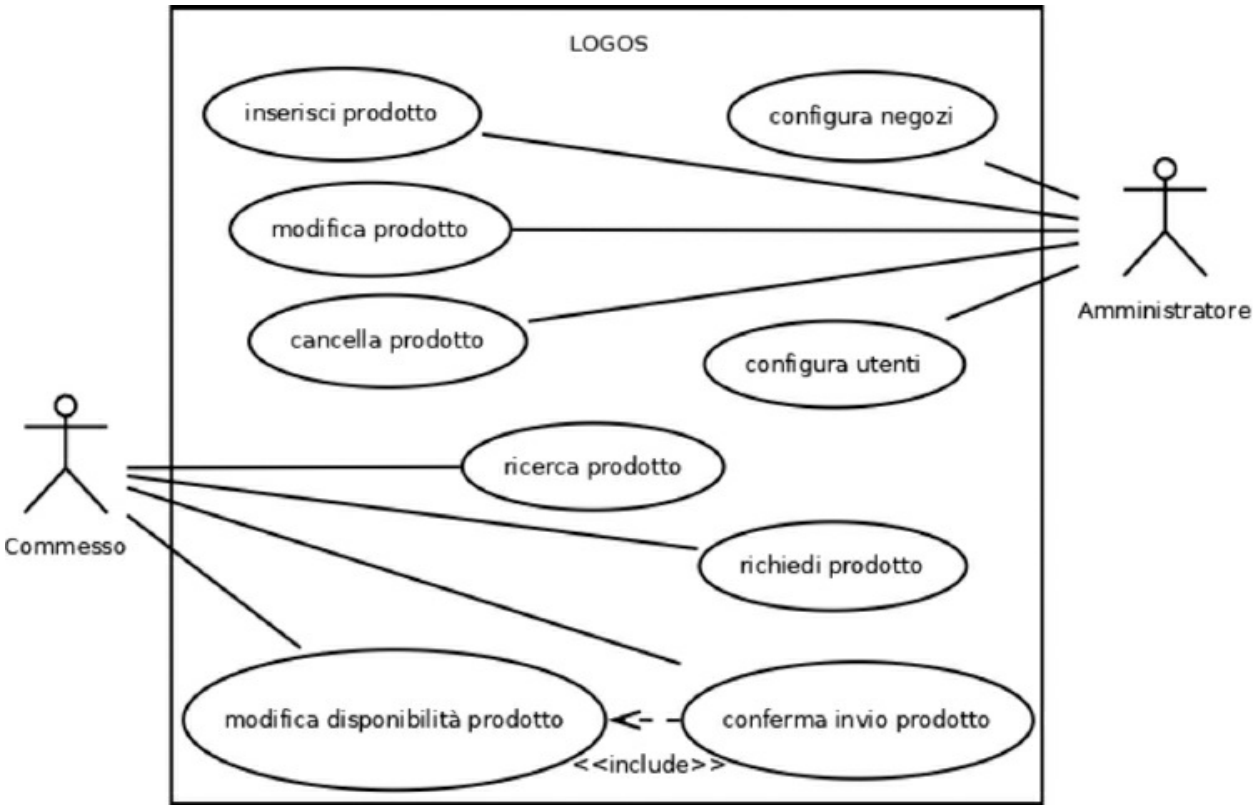


Figura 5.4 - Use case diagram di Logos.

Come è possibile aver ricavato tante funzionalità? Il processo che abbiamo seguito è il seguente. Ci siamo fatti varie domande coinvolgendo il cliente, ed abbiamo scoperto nuovi requisiti, e di volta in volta abbiamo aggiornato il diagramma, il quale, mostrando staticamente l'interazione tra utenti e sistema, ci ha aiutato a ragionare in maniera semplice. La nostra analisi è partita dai dubbi sul problem statement. Infatti in primis non ci convinceva la frase *deve permettere di ricercare tutti i prodotti vendibili, in tutti i negozi della catena mostrandone la disponibilità*. Se parliamo di disponibilità e ci sono cinque negozi, forse parliamo di **disponibilità per negozio.** Ed infatti il cliente ce lo ha confermato. E perché voler conoscere la disponibilità degli altri negozi? Il cliente ci ha spiegato che sarebbe molto utile poter chiedere l'invio di certi prodotti che scarseggiano in una sede alle altre sedi. Abbiamo proposto una soluzione basata sul seguente processo. Un commesso può cercare la disponibilità di un certo prodotto che manca o scarseggia nella sede di appartenenza usando la funzionalità ricerca prodotto. Può poi richiederlo ad un'altra sede utilizzando la funzionalità richiedi prodotto. Un commesso in un'altra sede riceverà questa richiesta tramite Logos e potrà confermare (o

meno) l'invio del prodotto richiesto all'altra sede tramite la funzionalità conferma invio prodotto. Tale funzionalità a sua volta diminuirà automaticamente la disponibilità del prodotto nella propria sede della quantità da inviare, usando automaticamente la funzionalità modifica disponibilità prodotto. Una volta ricevuto quanto richiesto, il commesso della sede richiedente dovrà aggiornare la disponibilità della propria sede utilizzando anch'egli la funzionalità modifica disponibilità prodotto. Un **vincolo** esplicitato nel diagramma con una **nota** (foglietto con angolo piegato) è che un commesso può aggiornare la disponibilità dei prodotti solo della propria sede. Questo processo è stato accettato dal cliente con entusiasmo. Quindi abbiamo aggiunto al diagramma tutti gli use case ed in un primo momento avevamo un unico attore: il Commesso, così come suggeriva il problem statement. Poi abbiamo realizzato che le funzionalità di inserimento prodotto, modifica prodotto, cancella prodotto, non possano essere assegnate ad un commesso, anche perché i prodotti sono condivisi da tutte le sedi e non da una in particolare. Quindi abbiamo proposto di aggiungere un nuovo attore chiamato Amministratore, che si deve occupare di definire i prodotti. Egli deve anche avere la possibilità di definire le sedi con i loro dettagli, e le utenze dei dipendenti (commessi) che dovranno autenticarsi nel sistema per poter eseguire operazioni. Con il diagramma in figura 5.4, è rappresentato il risultato della nostra analisi.

5.4.1.2 Scenari dei casi d'uso

Dopo aver individuato tutti i casi d'uso, la nostra analisi deve continuare andando ad esplicitare i cosiddetti **scenari degli use case.** Lo **scenario** per definizione è una sequenza di passi che descrivono l'interazione tra l'utenza ed il sistema. Si dice che lo scenario è un'istanza di un caso d'uso. Il **caso d'uso** infatti viene definito in UML come un insieme di scenari legati da un obiettivo comune per l'utente (noi avevamo identificato il caso d'uso con una funzionalità, ma questa è la definizione "ufficiale"). In definitiva il prossimo passo consiste nell'esplicitare tutti gli scenari possibili per ogni caso d'uso. Uno scenario non contiene condizioni, esiste lo **scenario standard** che è quello che ci si aspetta di eseguire se tutto va come deve andare, e poi eventuali condizioni danno luogo ad altri scenari. La descrizione degli scenari aiuta ad individuare tutti i punti critici del sistema senza scrivere neanche una

riga di codice. Questo passaggio si può tranquillamente eseguire con del testo. Per esempio, scendiamo nei dettagli dello use case Ricerca prodotto. Lo **scenario standard** potrebbe essere costituito dai seguenti passi:

1. il commesso ricerca un certo prodotto specificando tipo di prodotto="sciarpa", e marca="Columbia";
2. il sistema restituisce tre risultati con prodotti che corrispondono alla ricerca, visualizzando per ognuno di essi anche tutti i dettagli (id, prezzo, colore, etc.);
3. il commesso seleziona il prodotto desiderato con un click;
4. il sistema propone una schermata che riporta la disponibilità del prodotto per ogni sede, mostrando accanto ad ogni risultato un pulsante richiedi prodotto, ed in fondo alla pagina un pulsante torna alla ricerca.

Esistono decine di alternative a questo scenario e per ognuna conviene compilare un nuovo scenario. Per individuarle, partire dallo scenario principale e, per ogni punto, cercare un'alternativa. Per esempio ecco un possibile **secondo scenario:**

1. il commesso ricerca un certo prodotto specificando tipo di prodotto="sciarpa";
2. il sistema trova 505 risultati, ma mostra solo i primi dieci, paginando tutti gli altri.

Fermiamoci qui, perché già abbiamo trovato un risultato interessante, senza sforzi e in brevissimo tempo: se la ricerca produce troppi risultati vogliamo paginarli per rendere la vita dell'utente più semplice! Ci avremmo mai pensato se ci fossimo buttati sul codice? Proseguiamo con il **terzo scenario:**

1. il commesso ricerca un certo prodotto specificando id="741231";
2. il sistema, avendo trovato un unico risultato, propone direttamente la schermata di cui al punto 4 dello scenario standard.

Anche questa soluzione sembra molto interessante! Se il sistema trova un

unico risultato salta i passi 2 e 3 dello scenario principale. Anche in questo caso: ci saremmo mai arrivati senza descrivere gli scenari? La risposta pare scontata e l'analisi degli scenari pare assolutamente fondamentale per iniziare a programmare con le idee chiare. Continueremo negli esercizi la descrizione di altri scenari. È possibile utilizzare anche un activity diagram UML per specificare tutti gli scenari con un unico diagramma.

5.4.2 Architettura di alto livello

Parallelamente all'analisi degli use case, nel nostro processo un architetto potrebbe individuare e proporre i requisiti non funzionali dell'applicazione Logos. Per esempio con il deployment diagram raffigurato in figura 5.5, andiamo a definire una **soluzione architetturale di alto livello** (ovvero senza specificare troppi dettagli).

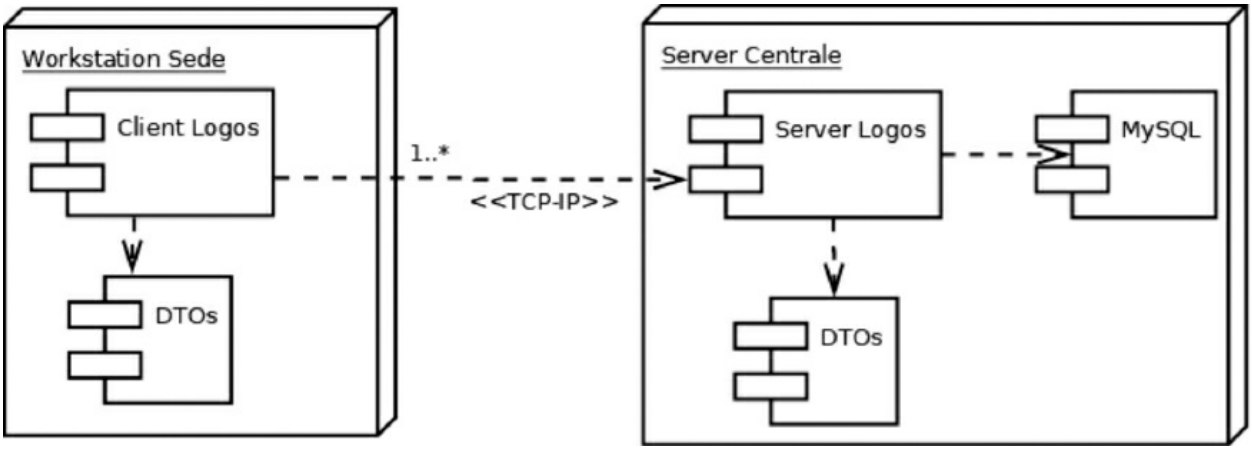


Figura 5.5 - High level deployment diagram di Logos.

In accordo con il cliente è stato di deciso di minimizzare i costi visto che i rischi di eventuali malfunzionamenti hardware e software non causerebbero particolari danni alla Panta Rei Clothing. Quindi si è scelto di avere un unico server centrale che sarà installato nella sede con la miglior connessione internet. Si tratterà di un'unica macchina dove girerà sia il modulo server di Logos, sia il database MySQL. La comunicazione con i client avverrà tramite protocollo TCP-IP. Cosa succede se per qualche ragione un client non riesce a comunicare con un server (requisito non funzionale)? È stato deciso di permettere al client di registrare alcuni dati localmente, che saranno sincronizzati con il server nel momento in cui la connessione tornerà

disponibile.

5.4.3 Key abstraction

Il passo successivo all'analisi dei casi d'uso è finalmente quello di definire quali sono le classi fondamentali che sicuramente saranno implementate nel nostro programma, le cosiddette **key abstraction** (in italiano **astrazioni chiave).** L'elenco di queste astrazioni chiave può essere ricavato a partire da un elenco di tutte le parole che sembrano avere un qualche senso per la nostra analisi. Questo elenco è chiamato elenco delle **classi candidate** (in inglese **candidate classes)**, e si ottiene semplicemente riportando tutte le parole che sembrano avere una qualche rilevanza all'interno dell'analisi degli use case e dal Deployment Diagram di alto livello. Non abbiamo completato l'analisi (l'abbiamo appena iniziata) quindi una lista completa non è facile da stilare, ma basandoci solo sul caso d'uso ricerca prodotto possiamo riportare le seguenti classi candidate: `commesso`, `ricerca`, `prodotto`, `sciarpa`, `risultato`, `paginazione`, `tipo di prodotto`, `identificativo (id)`, `prezzo`, `colore`, `disponibilità`, `schermata`, `client` e `server`. Come estrarre la sotto lista delle key abstraction? Semplicemente pensando che un'astrazione chiave deve avere già da ora una responsabilità chiara, altrimenti si scarta. Questo non significa che non diventerà più tardi una classe del sistema, ma semplicemente che non è un'astrazione chiave. Inoltre una buona regola dice che per quanto grande sia il progetto, la lista iniziale delle key abstraction non deve essere costituita da oltre sette entità. Nel nostro caso cercheremo di minimizzare ancora di più questo numero. In base a quanto detto escludiamo subito i concetti di `identificativo`, `prezzo`, `colore`, `tipo di prodotto`, ed anche `disponibilità`, che sembrano più che altro attributi della prima key abstraction `Prodotto` (la classe `Prodotto` è evidentemente essenziale per il business dell'applicativo). I concetti di `schermata` e `paginazione` sembrano appartenere più che altro all'interfaccia grafica del client, mentre sciarpa sembra un possibile valore dell'attributo `tipo di prodotto`. Il `commesso`, invece di un'astrazione chiave, è più che altro l'utente del sistema. Inoltre `ricerca` sembra più il nome di un metodo che il nome di una classe (anche se potrebbe anche adattarsi). Di fatto l'unica astrazione chiave sicura è `Prodotto`, visto che abbiamo esplorato parzialmente un unico caso d'uso. Anche se molto generiche per essere

considerate astrazioni chiave, per iniziare utilizzeremo anche le classi `Client` e `Server`. Ad esse assegneremo le responsabilità esecutive delle funzionalità dell'applicazione, dato che `Prodotto` rappresenta evidentemente un dato dell'applicazione. È possibile che queste classi poi si scindano in classi più specifiche andando avanti nel ciclo di sviluppo.

5.4.4 Progettazione con diagrammi di interazione

Il passo successivo è quindi quello di usare i diagrammi dinamici UML come il **sequence diagram** o il **collaboration diagram** (cfr. appendice L), per simulare tutti gli scenari realizzati assegnando le responsabilità delle azioni agli oggetti appositamente creati. Tali diagrammi hanno il compito di definire lo scambio dei messaggi tra oggetti che realizzerà il caso d'uso, e per dimostrare l'effettiva fattibilità dell'implementazione degli scenari utilizzando le classi individuate sinora. Nella figura 5.6 è stato rappresentato il terzo scenario che abbiamo analizzato con un **sequence diagram** (in italiano **diagramma di sequenza)**, che mostra come gruppi di oggetti interagiscono in un determinato lasso di tempo (nel nostro caso all'interno di uno scenario).

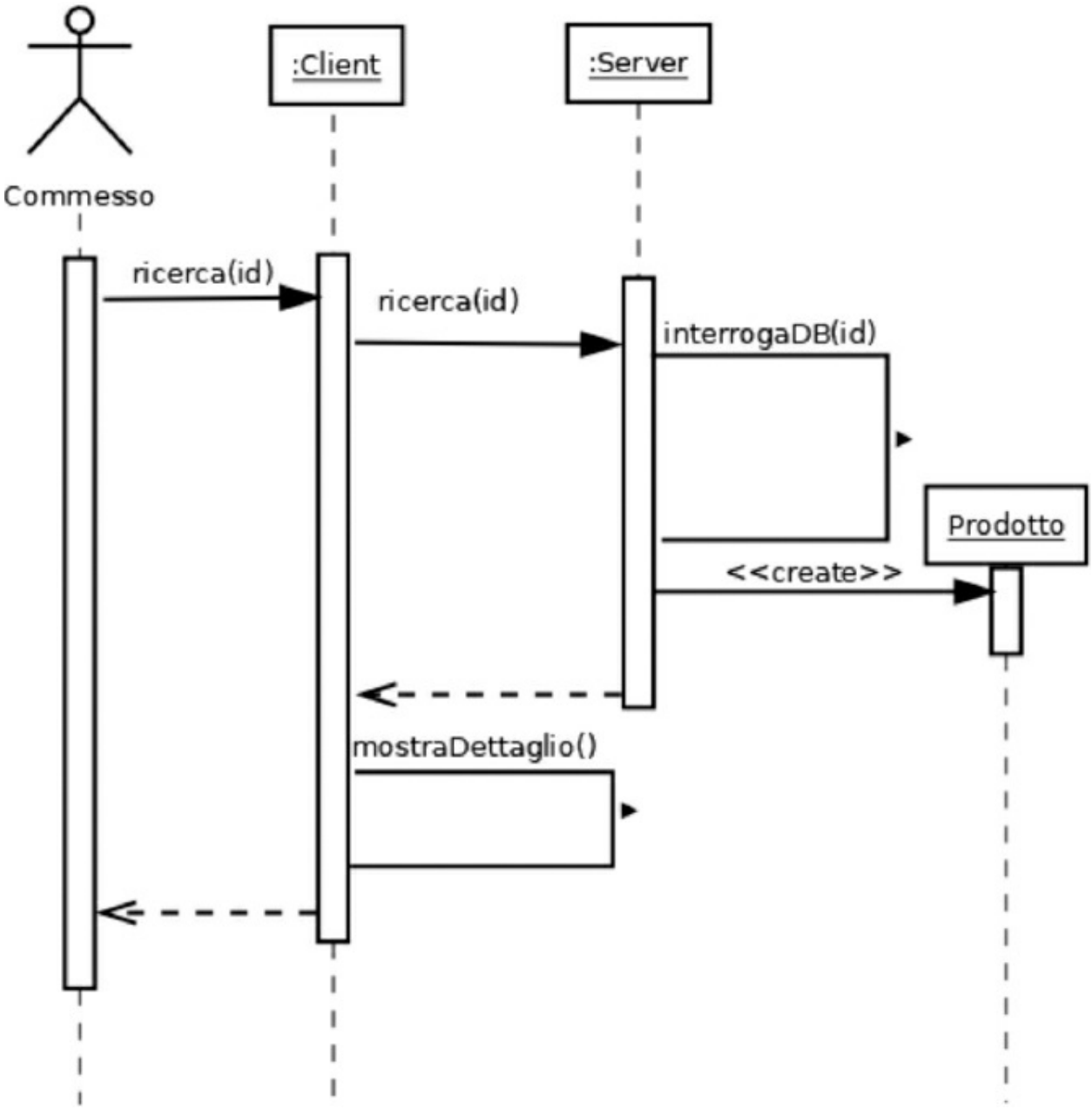


Figura 5.6 - Sequence diagram dello scenario 3 dello use case “ricerca prodotto”.

La sintassi di questo diagramma mostra nella parte superiore gli **oggetti** del sistema al runtime e un **attore** che dà il via allo scenario con una sua azione. I nomi degli oggetti hanno la sintassi: `nomeOggetto:nomeClasse`, con entrambe le parti opzionali (noi abbiamo omesso il `nomeOggetto`). Le linee tratteggiate che scendono dagli oggetti sono dette **linee di vita** (in inglese **life lines)** e rappresentano l’esistenza dell’oggetto. Si può notare come l’oggetto `Prodotto` inizia ad esistere nel momento in cui viene interrogato il database e

viene ritornato un risultato. Si noti che l'etichetta `<<create>>` indica che è stato chiamato il costruttore. Le stringhe racchiuse tra virgolette editoriali in UML si chiamano **stereotipi**, possono essere utilizzati su tutti i diagrammi e servono per fornire ulteriori informazioni a chi legge il diagramma. I rettangoli che coprono le linee di vita sono dette invece **linee di attività (activity lines)** e mostrano il periodo di tempo in cui un certo oggetto è attivo. Infine le frecce che congiungono gli oggetti sono detti **messaggi (messages)** e rappresentano i metodi, come è possibile intuire dalle loro etichette. I messaggi che tornano sullo stesso oggetto rappresentano metodi privati.

Riepilogo

JShell è un software della tipologia **REPL**, acronimo di read-evaluate-print loop che possiamo tradurre come ciclo leggi-valuta-scrivi. È possibile editare al suo interno sia snippet di codice, sia comandi JShell che vanno sempre eseguiti tramite il prefisso /. I comandi possono essere divisi in comandi esplorativi (**/list, /history, /help, /imports, /types, /vars** e **/methods**) e comandi esecutivi (**/set, /env, /exit, /drop, /edit, /open, /save, /reload** e **/reset**). Per quanto riguarda il codice che è possibile editare in JShell ci sono alcune **regole** che divergono dalla classica programmazione Java: è possibile dichiarare metodi ma non è obbligatorio dichiararne uno per scrivere uno statement; è possibile dichiarare classi ma non è obbligatorio dichiararne una per scrivere metodi o variabili; non è necessario utilizzare il simbolo ; per dichiarare statement in un'unica linea; non si possono dichiarare package e moduli; a parte `abstract` tutti gli altri modificatori fondamentali sono ignorati o non permessi per tutti i tipi non innestati; la parola chiave `synchronized` è sempre ignorata; dichiarare due volte la stessa variabile (o lo stesso metodo) causa la sovrascrittura della prima variabile dichiarata (o del primo metodo dichiarato); una variabile dichiarata e non inizializzata è automaticamente inizializzata al proprio valore nullo. JShell offre anche diversi strumenti ausiliari, come le **variabili implicite** che sono dichiarate automaticamente, il **forwarding reference** che permette di utilizzare strutture dati non ancora dichiarate, l'**auto-completamento** del codice, degli import e della dichiarazione delle variabili, la possibilità di aprire **JShell Edit Pad** (o

altri editor) e le **scorciatoie da tastiera.** Essendo eseguito da **riga di comando**, JShell usufruisce anche di strumenti come il copia-incolla, la ricerca, etc.

Gli **IDE** ci consentono, una volta che abbiamo preso confidenza, di sviluppare con maggiore velocità e correttezza. Tra i tanti vantaggi che offrono, ricordiamo la gestione automatica del **progetto**, un **editor** avanzato con varie funzionalità di **refactoring**, un **debugger** integrato che rappresenta uno strumento fondamentale per il programmatore, e l'**integrazione** con altri fondamentali software ausiliari per lo sviluppo come database, application server, version control system, ed ovviamente diversi framework.

Quando si lavora in una **realtà aziendale** ci si scontra con varie difficoltà, e purtroppo non basta sapere solo programmare. Bisogna conoscere l'**architettura** del progetto per orientarsi nel codice, ed avere consapevolezza di cosa stiamo sviluppando. A tal riguardo è possibile utilizzare diagrammi UML come i **component** e i **deployment diagrams.** Sono richieste diverse altre **conoscenze di base**, come quella relativa ai database, alle reti, all'XML, al web e così via. Esistono diverse **tipologie di programmi** come le applicazioni standalone, web, enterprise e mobile. In azienda le varie attività necessarie allo sviluppo sono svolte da diverse persone che hanno determinati **ruoli.** A seconda dell'organizzazione e della grandezza dell'azienda, capita spesso che alcune persone possano interpretare anche più di un ruolo. Una **metodologia** è composta da un **processo** (ovvero i passi da eseguire) e un **linguaggio di notazione (UML** che è considerato uno standard dalle più importanti metodologie). Abbiamo anche introdotto un **caso di studio** (con un breve problem statement) che da questo capitolo in poi sarà progressivamente sviluppato capitolo dopo capitolo. Ne abbiamo approfittato per introdurre un **processo** che definisce alcune semplici tecniche che ci permettono di iniziare un programma da zero. In particolare l'**analisi degli use case** ci consente di individuare quali sono le funzionalità dell'applicazione, e gli attori che le utilizzeranno, grazie anche all'aiuto del **diagramma degli use case.** Esplicitando gli **scenari** per ogni use case, riusciremo ad individuare tutti i punti critici del sistema prima ancora di iniziare a scrivere codice. Questo passo può essere eseguito con semplice **testo**, o anche con diagrammi dinamici come l'**activity diagram**, con il quale sarebbe possibile anche descrivere tutti gli scenari con un unico diagramma. Nel frattempo si può scegliere, anche in base alle esigenze e i vincoli del

cliente, un**'architettura di alto livello** per l'applicazione descrivendola con un **component diagram** innestato in un **deployment diagram.** Si può ottenere la lista delle **astrazioni chiave** del programma (ovvero le classi che sicuramente saranno implementate sin dall'inizio) estraendola dalla lista di tutte le **candidate classes** (lista composta da ogni termine che abbia una qualche rilevanza per l'analisi sinora fatta). Infine è possibile convalidare con dei **diagrammi di interazione** (di **sequence** o **collaboration diagram)** l'effettiva applicabilità degli scenari con le key abstraction individuate ed eventualmente aggiungere nuovi astrazioni.

> **Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.**

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper interagire con JShell per sperimentare, esplorare e scoprire Java (unità 5.1)	☐	
Avere una conoscenza delle potenzialità degli IDE (unità 5.2)	☐	
Comprendere le difficoltà di lavorare in una realtà aziendale (unità 5.3)	☐	
Saper utilizzare un processo che permetta di iniziare un programma da zero (unità 5.4)	☐	

Note:

Parte II

Object Orientation

La parte II è interamente dedicata al supporto che Java offre ai paradigmi della programmazione ad oggetti ed altri meccanismi correlati. Questa è forse la parte più importante di questo testo. Infatti dopo aver studiato i primi capitoli, anche un lettore neofita della programmazione, dovrebbe già essere in grado di scrivere del codice Java. Ma “smanettare” con un linguaggio di programmazione non è sufficiente, neanche quando si hanno grandi capacità. Come organizzare un programma partendo da zero? Quante classi bisogna creare? Come si devono chiamare queste classi? Che metodi dovranno definire? Come possiamo creare un programma capace di evolversi senza ogni volta ritoccare parti già scritte? A queste e a tante altre domande risponde la teoria dell’Object Orientation. I paradigmi dell’OO saranno presentati in maniera tale che il lettore impari ad apprezzarne l’utilità nella pratica. In più si è cercato di compiere un’operazione di schematizzazione degli argomenti abbastanza impegnativa, in particolare del polimorfismo, per facilitarne l’apprendimento. Affronteremo anche il meccanismo della gestione delle eccezioni e della progettazione per contratto con le asserzioni. Consigliamo lo studio di questa sezione a chiunque, anche a chi programma in Java da tempo.

Al termine di questa parte il lettore dovrebbe aver appreso le nozioni fondamentali dell’Object Orientation applicabili a Java, e quindi potrà accedere alla terza parte del libro riguardante le caratteristiche avanzate del linguaggio.

6

Incapsulamento e visibilità

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Comprendere le ragioni della nascita della programmazione ad oggetti (unità 6.1).
- ✔ Saper elencare i paradigmi ed i concetti fondamentali della programmazione ad oggetti (unità 6.2).
- ✔ Saper creare un diagramma delle classi di base (unità 6.2, 6.9).
- ✔ Saper definire ed utilizzare il concetto di astrazione e di riuso (unità 6.2).
- ✔ Comprendere l'utilizzo e l'utilità dell'incapsulamento (unità 6.3, 6.4, 6.5).
- ✔ Comprendere l'utilizzo e l'utilità del reference `this` (unità 6.3).
- ✔ Capire quando e come usare l'incapsulamento (unità 6.4, 6.5).
- ✔ Comprendere l'utilizzo e l'utilità dei package (unità 6.6, 6.7).
- ✔ Capire e saper utilizzare i modificatori d'accesso (unità 6.7).
- ✔ Capire e saper utilizzare il modificatore `static` (unità 6.8).
- ✔ Saper accennare al concetto di Design Pattern e comprendere l'implementazione del Singleton pattern (unità 6.8).

Questo è il primo capitolo che si occupa di approfondire il supporto offerto

da Java all'Object Orientation. In particolare verranno introdotti i primi paradigmi di questa scienza, come l'incapsulamento, il riuso e l'astrazione. Cercheremo di introdurre il lettore all'argomento partendo con l'esposizione delle ragioni storiche della nascita dell'Object Orientation. Il capitolo contiene anche altri concetti propedeutici allo studio dei prossimi capitoli.

6.1 Breve storia della programmazione ad oggetti

Scrivere un programma significa in qualche modo simulare su un computer concetti e modelli fisici e matematici. Nei suoi primi tempi, la programmazione era concepita come una serie di passi lineari. Invece di considerare lo scopo del programma nella sua interezza creandone un modello astratto, si cercava di arrivare alla soluzione del problema superando passaggi intermedi. Questo modello di programmazione orientato ai processi, con il passare del tempo, e con il conseguente aumento delle dimensioni dei programmi, ha manifestato apertamente i suoi difetti. Infatti, aumentando il numero delle variabili e delle interazioni da gestire tra esse, il programmatore in difficoltà iniziò a fare un uso sconsiderato degli strumenti che aveva a disposizione, in particolare delle **variabili globali** e del comando `goto`.

> **Il costrutto `goto` consentiva di far saltare il flusso di esecuzione del programma ad una riga arbitraria in qualsiasi momento. Le variabili globali invece, sono variabili visibili a tutte le parti del programma e quindi modificabili in qualsiasi momento e in qualsiasi situazione.**

L'utilizzo indiscriminato del costrutto `goto` e delle variabili globali però, ha spesso compromesso le funzionalità dei programmi. Infatti quando c'era bisogno di evolvere il software, o risolvere bachi, questi strumenti hanno sempre rappresentato scorciatoie troppo invitanti per non essere utilizzate. In questo modo, agli inizi degli anni settanta, per una certa programmazione funzionale fu coniato il termine dispregiativo "spaghetti code", dal momento che i programmi, crescendo in dimensioni, davano sempre più l'idea di

assomigliare ad una massa di pasta aggrovigliata.

La programmazione orientata agli oggetti nacque storicamente sin dagli anni sessanta con il linguaggio **Simula-67.** Non si trattava di un linguaggio orientato agli oggetti puro, ma con esso furono introdotti fondamentali concetti della programmazione quali le classi e l'ereditarietà. Fu sviluppato nel 1967 da Kristen Nygaard dell'università di Oslo e Ole Johan Dahl del Centro di Calcolo Norvegese e, a dispetto dell'importanza storica, non si può parlare di un vero e proprio successo presso il grande pubblico. Nei primi anni '70 nacque il linguaggio **SmallTalk**, sviluppato inizialmente da Alan Kay all'Università dello Utah e successivamente da Adele Goldberg e Daniel Ingalls dello Xerox Park, centro di ricerca di Palo Alto in California. SmallTalk si può considerare un linguaggio ad oggetti puro; introdusse l'incapsulamento e la release SmallTalk-80 ebbe anche un discreto successo negli Stati Uniti. A lungo andare però, sebbene considerato da molti come ideale ambiente di programmazione, rimase confinato (come Simula) nei centri di ricerca universitari di tutto il mondo, considerato come ambiente di studio più che di sviluppo. L'introduzione nel mondo della programmazione dei concetti di classe e di oggetto, che di fatto rendono i programmi più facilmente gestibili, non provocò quindi immediatamente una rivoluzione nell'informatica. Ciò fu dovuto al fatto che agli inizi degli anni Settanta ottenevano i maggiori successi linguaggi come il **C.** Un esempio su tutti: il sistema operativo **Unix**, tutt'oggi ancora utilizzatissimo, nacque proprio in quegli anni, ed il suo kernel (nucleo) era scritto in C. Immaginiamo lo scenario dell'epoca: linguaggi come il C, il Cobol ed il Fortran, erano utilizzati in maniera massiccia in molte applicazioni professionali con risultati ottimi e sempre più entusiasmanti. Critiche del tutto teoriche al paradigma procedurale, furono interpretate da molti semplicemente come fuori luogo, ma oggi sappiamo che più che altro si doveva parlare di critiche "fuori tempo"! Infatti, il sempre più frenetico progresso degli hardware, era ovviamente accompagnato da un'irrefrenabile evoluzione dei software che, utilizzando nuove risorse, dovevano gestire una sempre crescente quantità di variabili e di funzioni. I limiti del paradigma procedurale furono evidenti verso la fine degli anni '70, quando ormai la filosofia di creare software aveva ormai preso una strada definitiva. Il software doveva essere *estensibile*, per adattarsi alle nuove tecnologie che nascevano una dopo l'altra. Si iniziarono a rilasciare software in versioni successive (1.0, 1.1, 1.2, 2.0 etc.)

dette **release**, e questo ovviamente richiedeva un continuo aggiornamento del codice. Per apportare modifiche al software, linguaggi come il C, offrivano strumenti come le variabili globali e il comando `goto`, che come abbiamo detto si possono considerare ad alto rischio. Ecco che allora fu provvidenzialmente introdotta l'estensione del linguaggio C, realizzata da Bjarne Stroustrup, nota con il nome di **C++.** Questo nuovo linguaggio ha effettivamente rivoluzionato il mondo della programmazione. Fu scelto come linguaggio standard tra tanti linguaggi object oriented dalle grandi corporation (Microsoft, Borland, etc.), le quali iniziarono a produrre a loro volta tool di sviluppo che estendevano il C++ creandone una loro versione. Essendo un'estensione del C, un qualsiasi programma scritto in C deve poter essere compilato da un compilatore C++. Ciò, anche se ha favorito la migrazione in massa dei programmatori C verso il C++, si è rivelato anche uno dei limiti principali di quest'ultimo. Infatti il programmatore C che migrava a C++ poteva facilmente finire con lo scrivere soprattutto programmi funzionali poco orientati agli oggetti. Da qui l'idea di realizzare un nuovo linguaggio che doveva essere veramente orientato agli oggetti. Java propone uno stile di programmazione che quasi obbliga a programmare correttamente ad oggetti. Inoltre, rispetto al C++, sono stati eliminati tutti gli strumenti ambigui e pericolosi, come ad esempio il `goto`, l**'aritmetica dei puntatori** (strumento per interagire direttamente con la memoria, tanto potente quanto pericoloso) e, di fatto, per utilizzare un qualcosa che assomigli ad una variabile globale, ci si deve proprio impegnare. Possiamo concludere che, se il C++ ha il merito di aver fatto conoscere al grande pubblico la programmazione ad oggetti, Java ha il merito di averla fatta capire!

La **programmazione orientata agli oggetti** è una scienza, o meglio una filosofia adattabile alla programmazione. Essa si basa su concetti esistenti nel mondo reale, con i quali abbiamo a che fare ogni giorno. È già stato fatto notare al lettore che gli esseri umani posseggono da sempre i concetti di **classe** e di **oggetto.** L'**astrazione** degli oggetti reali in classi ci fa superare la complessità della realtà. In questo modo possiamo osservare oggetti completamente differenti, riconoscendo in loro caratteristiche e funzionalità che li accomunano, e quindi associarli ad una stessa classe. Per esempio, sebbene completamente diversi, un sassofono ed un pianoforte appartengono entrambi alla classe degli strumenti musicali. La programmazione ad oggetti

inoltre, utilizzando il concetto di **incapsulamento**, rende i programmi composti da classi che nascondono i dettagli di implementazione dietro ad interfacce pubbliche, le quali permettono la comunicazione tra gli oggetti stessi che fanno parte del sistema. È favorito il **riuso** di codice già scritto anche grazie a concetti quali l'**ereditarietà** ed il **polimorfismo**, che saranno presentati al lettore nei prossimi capitoli.

C'è da sottolineare che con Java 8 è stato fatto un passo rivoluzionario. Dopo quasi vent'anni sono stati introdotti costrutti tipici della **programmazione funzionale** per permettere al programmatore di avere maggiore libertà, e avere un linguaggio più potente. Tuttavia Java rimane e rimarrà per sempre un linguaggio soprattutto object oriented.

6.2 I paradigmi della programmazione ad oggetti

Dalla lettura del paragrafo precedente quindi, si può evincere che la programmazione ad oggetti ci permetterà di creare programmi più semplici da evolvere e manutenere, che riusano parti di codice ben definite, e che facilitano l'interazione tra il programmatore e il codice. I vantaggi non si limitano solo a queste caratteristiche, e scopriremo nuovi benefici durante la lettura dei prossimi capitoli. Ciò che caratterizza un linguaggio orientato agli oggetti è il supporto che esso offre ai cosiddetti **paradigmi della programmazione ad oggetti:**

- **Incapsulamento**
- **Ereditarietà**
- **Polimorfismo**

A differenza di altri linguaggi di programmazione orientati agli oggetti, Java definisce in modo estremamente chiaro i concetti appena accennati. Anche programmatori che si ritengono esperti di altri linguaggi orientati agli oggetti come il C++, studiando Java potrebbero interpretare diversamente alcuni concetti che prima magari risultavano poco chiari.

Nel presente e nei prossimi capitoli introdurremo i tre paradigmi in questione. Segnaliamo al lettore che non tutti i testi parlano di tre paradigmi.

In effetti si dovrebbero considerare paradigmi della programmazione ad oggetti anche l'astrazione e il riuso. Questi ultimi due sono però considerati secondari rispetto agli altri, non perché meno potenti e utili, ma perché non sono specifici della programmazione orientata agli oggetti. Infatti l'astrazione e il riuso sono concetti che appartengono anche alla programmazione funzionale.

> **I paradigmi fondamentali dell'Object Orientation sono tanti, ma non è questa la sede ideale dove approfondire il discorso.**

6.2.1 Astrazione

L'**astrazione** potrebbe definirsi come "l'arte di sapersi concentrare solo sui dettagli veramente essenziali nella descrizione di un'entità". L'astrazione è un concetto chiarissimo a tutti noi, dal momento che lo utilizziamo in ogni istante della nostra vita. Per esempio, mentre state leggendo questo manuale, vi state concentrando sull'apprenderne correttamente i contenuti, senza badare troppo alla forma, ai colori, allo stile e tutti particolari fisici e teorici che compongono la pagina che state visualizzando (o almeno lo speriamo!). Per formalizzare un discorso altrimenti troppo "astratto", potremmo parlare di almeno tre livelli di astrazione per quanto riguarda la sua implementazione nella programmazione ad oggetti:

- astrazione funzionale
- astrazione dei dati
- astrazione del sistema

Adoperiamo l'**astrazione funzionale** ogni volta che implementiamo un metodo. Infatti, tramite un metodo, riusciamo a portare all'interno di un'applicazione un concetto dinamico, sinonimo di azione, funzione. Per scrivere un metodo ci dovremmo limitare alla sua implementazione più robusta e chiara possibile. In questo modo avremo la possibilità di invocare quel metodo ottenendo il risultato voluto, senza dover tener presente

l’implementazione del metodo stesso.

> **Lo stesso concetto è valido anche nella programmazione funzionale, grazie alle funzioni.**

Adoperiamo l’**astrazione dei dati** ogni volta che definiamo una classe, raccogliendo in essa solo le caratteristiche e le funzionalità essenziali degli oggetti che essa deve definire nel contesto in cui ci si trova.

> **Potremmo dire che l’astrazione dei dati “contiene” l’astrazione funzionale.**

Adoperiamo l’**astrazione del sistema** ogni volta che definiamo un’applicazione nei termini delle classi essenziali che devono soddisfare gli scopi dell’applicazione stessa.

> **Potremmo affermare che l’astrazione del sistema “contiene” l’astrazione dei dati, e per la proprietà transitiva, l’astrazione funzionale.**

Implementare l’astrazione in Java, a volte significa semplicemente saper trovare un nome appropriato alle entità (metodi, variabili, classi, etc.) che definiamo. Esempi di astrazione corretta sono: una classe `Poeta` che definisce il metodo `scriviPoesia()`, un metodo `stampaRisultato()` che stampa un risultato e fa solo quello, un package che si chiama `cinema.genere` che definisce le classi `Drammatico, Fantasy, Thriller, Commedia`. Esempi di cattiva astrazione invece potrebbero essere una classe `Poeta` che oltre al metodo `scriviPoesia()` definisce il metodo `stampaRisultato()` (i poeti non stampano risultati), il metodo `stampaRisultato()` che compie un complicato algoritmo prima di stampare il risultato (in quel caso sarebbe consigliabile creare due metodi separati, uno che esegue l’algoritmo, ed un altro per la

stampa), e un package `cinema.genere` che definisce anche la classe `Sceneggiatura` (che non è un genere e quindi non dovrebbe trovarsi in quel package).

6.2.2 Riuso

Con il termine **riuso** ci riferiamo principalmente al concetto di riuso che abbiamo nel mondo reale, che riportato nel mondo Java viene tradotto nel riuso di elementi di programmazione come classi e metodi, ma anche di concetti e processi che solitamente utilizziamo quando sviluppiamo un programma. Il **riuso** è da considerarsi una conseguenza dell'astrazione e degli altri paradigmi della programmazione ad oggetti (incapsulamento, ereditarietà e polimorfismo). Questi infatti favoriscono il riuso intrinsecamente.

Possiamo anche in questo caso parlare di tre livelli di astrazione per quanto riguarda la sua implementazione nella programmazione ad oggetti:

- riuso funzionale
- riuso dei dati
- riuso concettuale

Utilizziamo il **riuso funzionale** ogni volta che riusciamo a creare un metodo ben astratto che viene invocato in più parti dal nostro codice. Questo significa soprattutto non duplicare codice. Quando duplichiamo codice con un copia-incolla da un metodo ad un altro, probabilmente stiamo sbagliando qualcosa. Dovremmo prenderci il tempo per valutare la creazione di un metodo che sia chiamato da entrambi i metodi contenente il codice in comune.

> **Il riuso è infatti un paradigma valido anche per la programmazione funzionale.**

Nella libreria standard di Java sono presenti migliaia di metodi che possiamo riutilizzare in tante applicazioni, anche completamente differenti tra loro.

Utilizziamo il **riuso dei dati** ogni volta che riusciamo a creare una classe

ben astratta e con una interfaccia di metodi pubblici usabile, che viene istanziata in più parti dal nostro codice. Ricade nel caso del riuso dei dati anche il caso in cui viene creato un **componente software**, ovvero un insieme di classi con un'interfaccia ben definita, che mette a disposizione funzionalità eseguibili indipendentemente dal contesto.

Utilizziamo invece il **riuso concettuale** quando riusciamo a riutilizzare una tecnica, un algoritmo, un approccio al problema, una soluzione progettuale o architetturale già nota.

6.2.3 UML: Diagramma delle classi

Alla fine del capitolo 4 abbiamo introdotto il linguaggio di notazione UML. Per avere una nuova prospettiva di visualizzazione del codice, introduciamo una nuova tipologia di diagramma: **il diagramma delle classi** (in inglese **class diagram).** Come già asserito utilizzeremo UML senza la pretesa di creare precise specifiche per programmare. Ci limiteremo semplicemente a creare degli "schizzi" per ottenere informazioni che leggendo direttamente il codice non sarebbero altrettanto chiare.

La notazione di classe in UML è molto intuitiva, un rettangolo diviso in tre sezioni: nella prima viene scritto il nome della classe, nella seconda la lista degli attributi, nella terza l'elenco dei metodi.

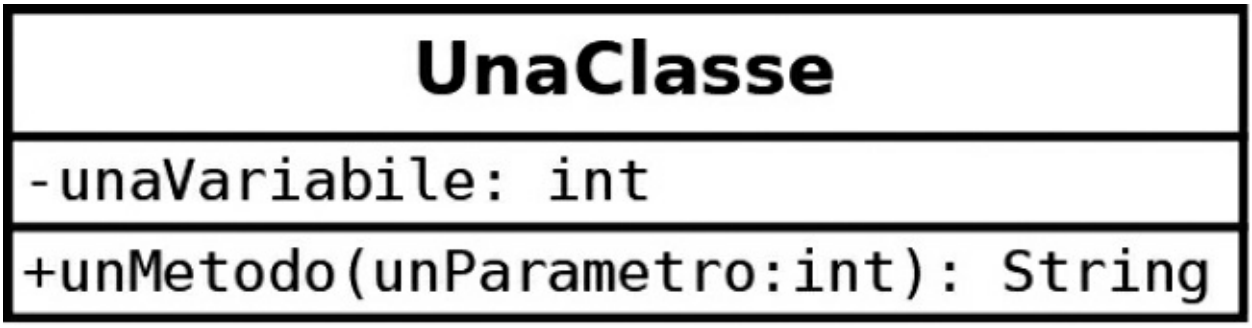


Figura 6.1 - Class Diagram "Classe Cliente".

UML è un linguaggio di modellazione indipendente dal linguaggio di programmazione che vuole rappresentare. Questo significa che non dobbiamo aspettarci attributi e metodi descritti con la sintassi Java. Come è possibile vedere dalla figura 6.1 gli attributi vengono definiti con la seguente sintassi UML:

```
  simbolo_di_visibilità nomeAttributo :
tipoAttributo
```

Sempre osservando la figura 6.1, si può notare che il simbolo di visibilità della variabile `unaVariabile` che abbiamo definito nel diagramma è `-`, che equivale al modificatore `private`. Quindi la sintassi:

```
  - unaVariabile :  int
```

in UML, equivale alla sintassi:

```
  private int unaVariabile;
```

in Java.

Per quanto riguarda i metodi la sintassi UML:

```
  simbolo_di_visibilità
nomeMetodo(listaDiParametri) :  tipoDiRitorno
```

dove con `listaDiParametri` intendiamo una lista di parametri che usa come separatore il simbolo della virgola, dove ogni parametro utilizza la sintassi:

```
  nomeParametro :  tipoParametro
```

Facendo riferimento alla figura 6.1, è facile intuire che il simbolo `+` corrisponde al modificatore `public`. Quindi la sintassi:

```
  + unMetodo(unParametro:int) :  String
```

in UML, equivale alla sintassi:

```
  public String unMetodo(int unParametro)
```

in Java. Notare che per i metodi in UML non viene specificato il corpo. Il diagramma delle classi infatti permette la definizione dell'**interfaccia pubblica** degli oggetti che vogliamo creare, nascondendo l'**implementazione interna.** Il perché convenga separare l'interfaccia

pubblica dall'implementazione interna, è argomento del prossimo paragrafo.

> **È possibile definire tanti altri elementi nel diagramma delle classi usando a volte una sintassi abbastanza criptica. Con la sintassi base si riescono a coprire comunque quasi la totalità delle esigenze dello sviluppo. Introdurremo nuovi elementi di sintassi ogni volta che ci saranno utili.**

6.3 Incapsulamento

L'**incapsulamento** è la chiave della programmazione orientata agli oggetti. Tramite esso, una classe riesce ad acquisire caratteristiche di robustezza, indipendenza e riusabilità. Inoltre la sua manutenzione risulterà più semplice al programmatore. Una qualsiasi classe è essenzialmente costituita da dati e metodi. La filosofia dell'incapsulamento è semplice. Essa si basa sull'accesso controllato ai dati mediante metodi che possono prevenirne l'usura e la non correttezza. A livello di implementazione, ciò si traduce semplicemente nel dichiarare privati gli attributi di una classe e quindi inaccessibili fuori dalla classe stessa. Per tale scopo introdurremo un nuovo modificatore: `private`. L'accesso ai **dati privati** potrà essere fornito da un**'interfaccia pubblica** costituita da metodi dichiarati `public` e quindi accessibili da altre classi. In questo modo tali metodi potrebbero ad esempio permettere di realizzare controlli prima di confermare l'accesso ai dati privati. Se l'incapsulamento è gestito in maniera intelligente le nostre classi potranno essere utilizzate nel modo migliore e più a lungo, giacché le modifiche e le revisioni potranno riguardare solamente parti di codice non visibili all'esterno. Se volessimo fare un esempio basandoci sulla realtà che ci circonda potremmo prendere in considerazione un telefono. La maggior parte degli utenti, infatti, sa utilizzare il telefono, ma ne ignora il funzionamento interno. Chiunque può scegliere un contatto dalla rubrica o comporre un numero telefonico e conversare con un'altra persona, ma pochi conoscono in dettaglio la sequenza dei processi scatenati da queste poche, semplici azioni. Evidentemente, per utilizzare il telefono, non è necessario prendere una laurea in ingegneria delle telecomunicazioni: basta conoscere la sua interfaccia pubblica (costituita dal

monitor o dalla tastiera del telefono, l'altoparlante e il microfono), non la sua implementazione interna. Passiamo ad un esempio di codice. Supponiamo di voler scrivere un'applicazione che utilizza la seguente classe, la quale astrae in maniera semplice il concetto di data:

```
public class Data {
    public int giorno;
    public int mese;
    public int anno;
}
```

Come può utilizzare la nostra applicazione tale astrazione? Ogni volta che serve un oggetto `Data`, il codice da scrivere sarà simile al seguente:

(**Codice 6.1**)

```
...
Data unaData = new Data();
unaData.giorno = 14;
unaData.mese = 4;
unaData.anno = 2004;
...
```

Qual è il problema? Non è raro che i valori delle variabili dell'oggetto debbano essere impostati al runtime in maniera dinamica, probabilmente dall'utente. Supponiamo che la nostra applicazione permetta all'utente di inserire la sua data di nascita, magari mediante un'interfaccia grafica. In tal caso il codice da scrivere sarà simile al seguente:

(**Codice 6.2**)

```
...
Data unaData = new Data();
unaData.giorno = interfaccia.dammiGiornoInserito();
unaData.mese = interfaccia.dammiMeseInserito();
unaData.anno = interfaccia.dammiAnnoInserito();
...
```

dove i metodi `dammiGiornoInserito()`, `dammiMeseInserito()` e `dammiAnnoInserito()` dell'oggetto `interfaccia` restituiscono un intero inserito dall'utente mediante l'applicazione. Supponiamo che l'utente abbia

inserito rispettivamente i valori `32` per il giorno, `13` per il mese e `1800` per l'anno; ecco che i problemi del codice iniziano a risultare evidenti. Come è possibile evitare definitivamente problemi come questo per la classe `Data`? Elenchiamo alcune possibili soluzioni.

1. Si potrebbe limitare la possibilità degli inserimenti sull'interfaccia grafica all'utente. Il problema sarebbe risolto, ma solo nel caso che l'impostazione della data avvenga sempre e comunque tramite l'interfaccia grafica. Inoltre il problema sarà risolto solo per quella particolare funzionalità e quindi non in maniera definitiva. Ma se volessimo riutilizzare in un'altra funzione la classe `Data` (il riuso dovrebbe essere un paradigma fondamentale della programmazione ad oggetti), senza riutilizzare la stessa interfaccia grafica, saremmo costretti a scrivere nuovamente del codice che gestisce il problema.
2. Potremmo delegare al codice dei metodi `dammiGiornoInserito()`, `dammiMeseInserito()` e `dammiAnnoInserito()` dell'oggetto `interfaccia` i controlli necessari alla giusta impostazione della data. Ma anche in questo caso rimarrebbero tutti i problemi esposti per il punto precedente.
3. Utilizzare l'incapsulamento modificando la classe `Data` nel modo seguente:

(**Codice 6.3**)

```
public class Data {
    private int giorno;
    private int mese;
    private int anno;

    public void setGiorno(int g) {
         if (g > 0 && g <= 31) {
             giorno = g;
         }
         else {
             System.out.println("Giorno non valido");
         }
    }

    public int getGiorno() {
```

```
            return giorno;
         }

        public void setMese(int m) {
             if (m > 0 && m <= 12) {
                 mese = m;
             }
             else {
                 System.out.println("Mese non valido");
             }
        }

        public int getMese() {
            return mese;
        }

        public void setAnno(int a) {
            anno = a;
        }

        public int getAnno() {
            return anno;
        }
    }
```

Implementare l'incapsulamento con codice Java consiste il più delle volte nel dichiarare i dati privati e fornire alla classe metodi pubblici di tipo **set** e **get** per accedervi rispettivamente in scrittura e lettura.

> **Questi metodi solitamente seguono una convenzione che è utilizzata anche nella libreria standard. Se abbiamo una variabile privata dovremmo chiamare questi metodi con la sintassi** `setNomeVariabile()` **e** `getNomeVariabile()`**. Esiste un'unica eccezione a questa convenzione quando abbiamo a che fare con una variabile di tipo** `boolean`**. In questo caso il metodo** `getNomeVariabile()` **va chiamato** `isNomeVariabile()`.

Quindi, anche se all'inizio potrà sembrare noioso (la seconda versione della

classe `Data` è nettamente più estesa della prima), implementare l'incapsulamento non richiede grossa inventiva da parte dello sviluppatore. Inoltre tutti gli IDE (anche EJE), offrono strumenti per la generazione automatica dei metodi set e get. Nella figura 6.2 sono rappresentate con UML la versione non incapsulata e la versione incapsulata della classe `Data`.

Possiamo notare in questo diagramma la notazione di **commento.** Un commento in UML è accettato in qualsiasi diagramma, e serve appunto per commentare qualsiasi cosa si voglia.

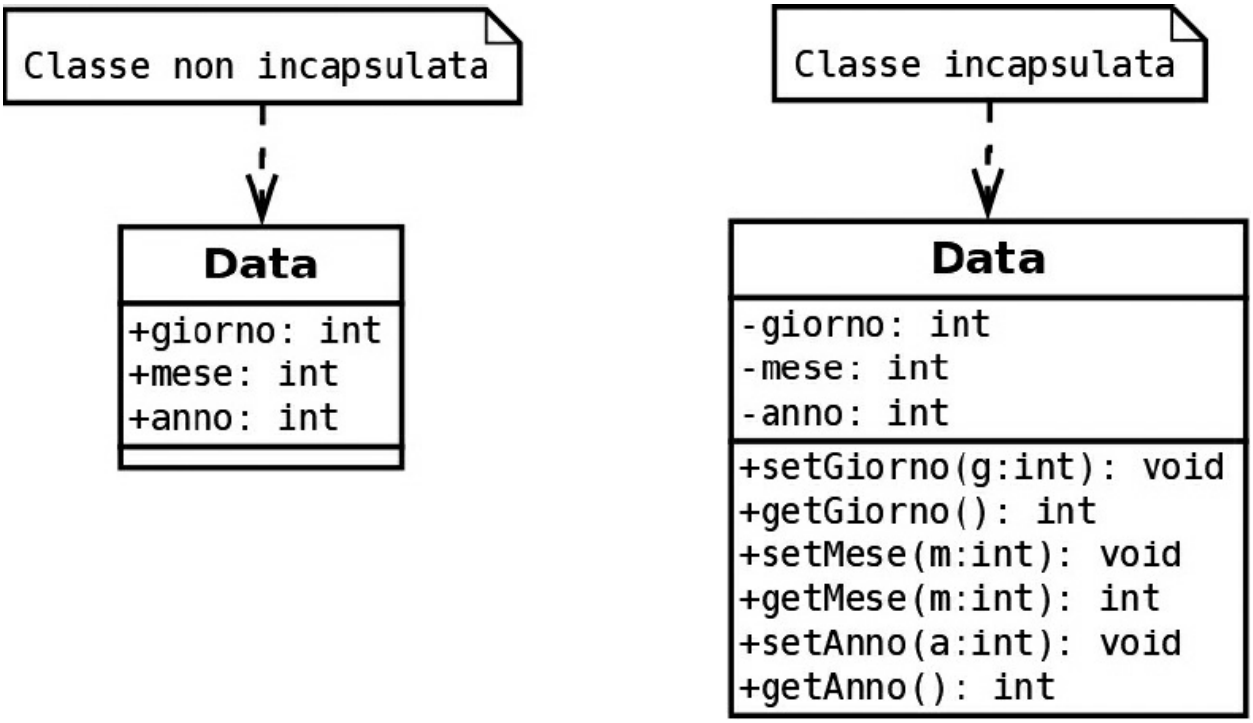


Figura 6.2 - Class Diagram "Classe Data".

Nel nostro caso abbiamo semplicemente specificato informazioni sulle classi definite. Ma è possibile scrivere all'interno dei commenti algoritmi, requisiti, codici, descrizioni, etc. Notare che il commento è unito all'elemento a cui è relazionato tramite una freccia tratteggiata. Tale freccia tratteggiata è detta **dipendenza**, ed implica dipendenza dalla fonte alla destinazione (nel nostro caso dal commento alla classe, perché il commento senza classe non avrebbe avuto senso). Anche le dipendenze, come i commenti, possono essere utilizzate su tutti i tipi di diagrammi UML.

Per ora possiamo sottolineare il valore di documentazione del diagramma delle classi, che ci mostra una versione incapsulata di una classe senza

specificare i dettagli. Questo, soprattutto in fase di progettazione della classe, può darci molti vantaggi. La visione grafica di una classe infatti, ci può aiutare a pensare alle classi più facilmente rispetto a guardare del codice Java. Quando leggiamo il codice di una classe è facile perdersi nei dettagli implementativi.

Cerchiamo ora di chiarire il perché sia così importante utilizzare l'incapsulamento. Nel momento in cui abbiamo dichiarato i dati privati, per la definizione del modificatore `private` essi non saranno più accessibili mediante l'operatore dot, a meno che il codice che vuole accedere al dato privato non si trovi nella classe che lo ha dichiarato. Questo implica che il codice 6.1 e il codice 6.2 produrrebbero un errore in compilazione in quanto tenterebbero di assegnare valori a variabili non visibili in quel contesto (classi diverse dalla classe `Data`). I codici 6.1 e 6.2 devono essere rispettivamente sostituiti con i seguenti:

(**Codice 6.1.bis**)

```
...
Data unaData = new Data();
unaData.setGiorno(14);
unaData.setMese(4);
unaData.setAnno(2004);
...
```

(**Codice 6.2.bis**)

```
...
Data unaData = new Data();
unaData.setGiorno(interfaccia.dammiGiornoInserito());
unaData.setMese(interfaccia.dammiMeseInserito());
unaData.setAnno(interfaccia.dammiAnnoInserito());
...
```

Ovvero, implementando l'incapsulamento, per sfruttare i dati privati dell'oggetto `Data`, saremo costretti ad utilizzare l'interfaccia pubblica dell'oggetto costituita dai metodi pubblici set e get, così come quando vogliamo utilizzare un telefono siamo costretti ad utilizzare l'interfaccia pubblica costituita dai tasti e dalla cornetta. Infatti i metodi set e get hanno implementazioni che si trovano internamente alla classe `Data` e quindi

possono accedere ai dati privati. Inoltre, nel codice 6.3, si può notare che, per esempio, il metodo `setGiorno()` imposta la variabile `giorno` con il parametro che gli viene passato se risulta compresa tra `1` e `31`, altrimenti stampa un messaggio di errore. Quindi, a priori, ogni oggetto `Data` funziona correttamente! Questo implica maggiori opportunità di riuso e robustezza del codice. Altro grande vantaggio: il codice è molto più facile da manutenere e si adatta ai cambiamenti. Per esempio, il lettore avrà sicuramente notato che il codice 6.3 risolve solo in parte i problemi della classe `Data`. Infatti permetterebbe l'impostazione del giorno al valore `31`, anche se la variabile mese per esempio vale `4` che equivale al mese di aprile che ha solo 30 giorni. E nemmeno risolve il problema che febbraio ha solo 28 giorni e negli anni bisestili 29. La buona notizia è che possiamo far evolvere la classe `Data`, introducendo tutte le migliorie che vogliamo all'interno del codice 6.3, ma non dovremmo cambiare una riga per i codici 6.1 bis e 6.2 bis. Per esempio, se il metodo `setGiorno()`, viene cambiato nel modo seguente:

```
public void setGiorno(int g) {
    if (g > 0 && g <= 31 && mese != 4) {
        giorno = g;
    }
    else {
        System.out.println("Giorno non valido");
    }
}
```

bisognerà ricompilare solo la classe `Data`, ma i codici 6.1.bis e 6.2.bis rimarranno inalterati.

Al lettore dovrebbe ora risultare chiara l'utilità dei metodi set che da ora in poi chiameremo **mutator methods**, ovvero metodi che mutano (il valore della variabile). Potrebbe però avere ancora qualche riserva sui metodi get che da adesso chiameremo **accessor methods** ovvero metodi d'accesso.

> **A volte ci si riferisce a questi metodi anche come metodi "setter" e "getter".**

Con un paio di esempi dovremmo fugare eventuali dubbi.

Supponiamo di volere aggiungere, al codice 6.2 bis, una verifica dell'effettivo successo dell'impostazione dei dati dell'oggetto `unaData`, stampandone i dati a video. Dal momento in cui:

```
System.out.println(unaData.giorno);
```

restituirà un errore in compilazione, e:

```
System.out.println(unaData.setGiorno());
```

non ha senso perché il tipo di ritorno del metodo `setGiorno()` è `void`, appare evidente che l'unica soluzione rimane:

```
System.out.println(unaData.getGiorno());
```

Inoltre anche un accessor method potrebbe eseguire controlli come un mutator method.

Per esempio, nella seguente classe l'accessor method gestisce l'accesso ad un conto bancario personale, mediante l'inserimento di un codice segreto:

```
public class ContoBancario {
    private String contoBancario = "5000000 Euro";
    private int codice = 1234;
    private int codiceInserito;

    public void setCodiceInserito(int cod) {
        codiceInserito = cod;
    }

    public int getCodiceInserito() {
        return codiceInserito;
    }

    public String getContoBancario() {

        if (codiceInserito == codice) {
            return contoBancario;
        }
        else {
            return "codice errato!!!";
        }
     }
     //...
```

```
}
```

6.3.1 Incapsulamento funzionale

Sino ad ora abbiamo visto esempi di incapsulamento abbastanza classici, dove nascondevamo all'interno delle classi gli attributi mediante il modificatore `private`. Nulla ci vieta di utilizzare `private`, anche come modificatore di metodi, ottenendo così un **incapsulamento funzionale.** Un metodo privato infatti, potrà essere invocato solo da un metodo definito nella stessa classe, che potrebbe a sua volta essere dichiarato pubblico. Per esempio la classe `ContoBancario` definita precedentemente, potrebbe evolversi nel seguente modo:

```
public class ContoBancario {
    //...
    public String getContoBancario(int codiceDaTestare) {
        return controllaCodice(codiceDaTestare);
    }

    private String controllaCodice(int codiceDaTestare) {
        if (codiceInserito == codiceDaTestare) {
            return contoBancario;
        }
        else {
            return "codice errato!!!";
        }
    }
}
```

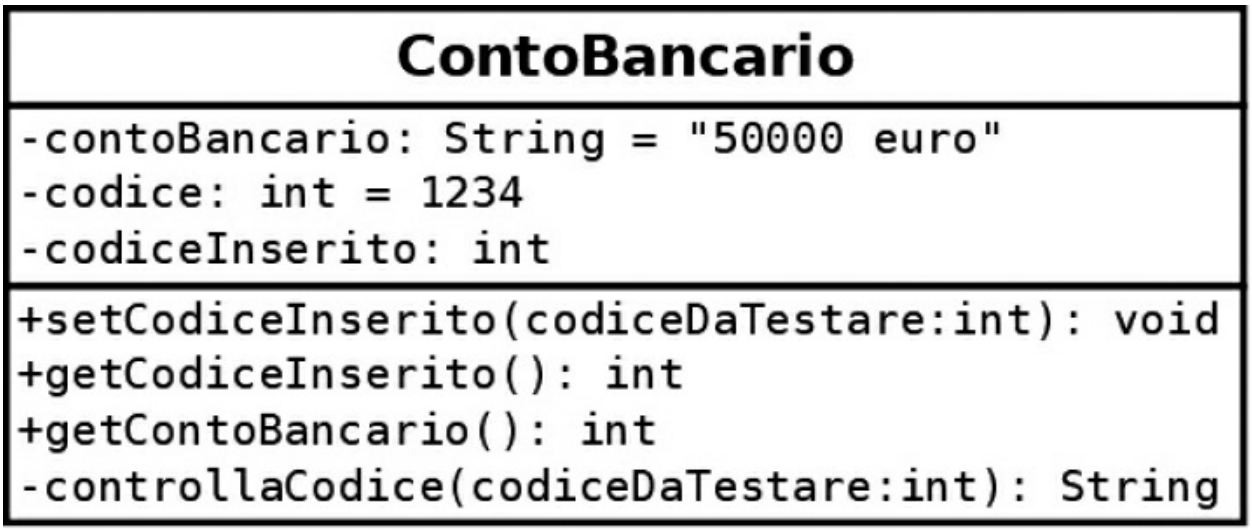


Figura 6.3 - Class Diagram “Classe ContoBancario”.

Ciò favorirebbe il riuso di codice in quanto, introducendo nuovi metodi (come probabilmente accadrà in un progetto che viene manutenuto), questi potrebbero riusare (riuso funzionale) il metodo `controllaCodice()`.

Nella figura 6.3 è rappresentata la classe `ContoBancario`. Notare come la notazione di classe riesce a rappresentare chiaramente il concetto di classe senza specificare i dettagli.

6.3.2 Incapsulamento e riuso

Solitamente si pensa che un membro di una classe dichiarato `private` diventi “inaccessibile da altre classi”. Questa frase è ragionevole per quanto riguarda l’ambito della compilazione, dove la dichiarazione delle classi è il problema da superare. Ma se ci spostiamo nell’ambito della Java Virtual Machine dove, come abbiamo detto i protagonisti assoluti non sono le classi ma gli oggetti, dobbiamo rivalutare l’affermazione precedente. L’incapsulamento infatti permetterà a due oggetti istanziati dalla stessa classe di accedere in “modo pubblico” ai rispettivi membri privati. Consideriamo la classe `Dipendente` rappresentata in UML nella figura 6.4:

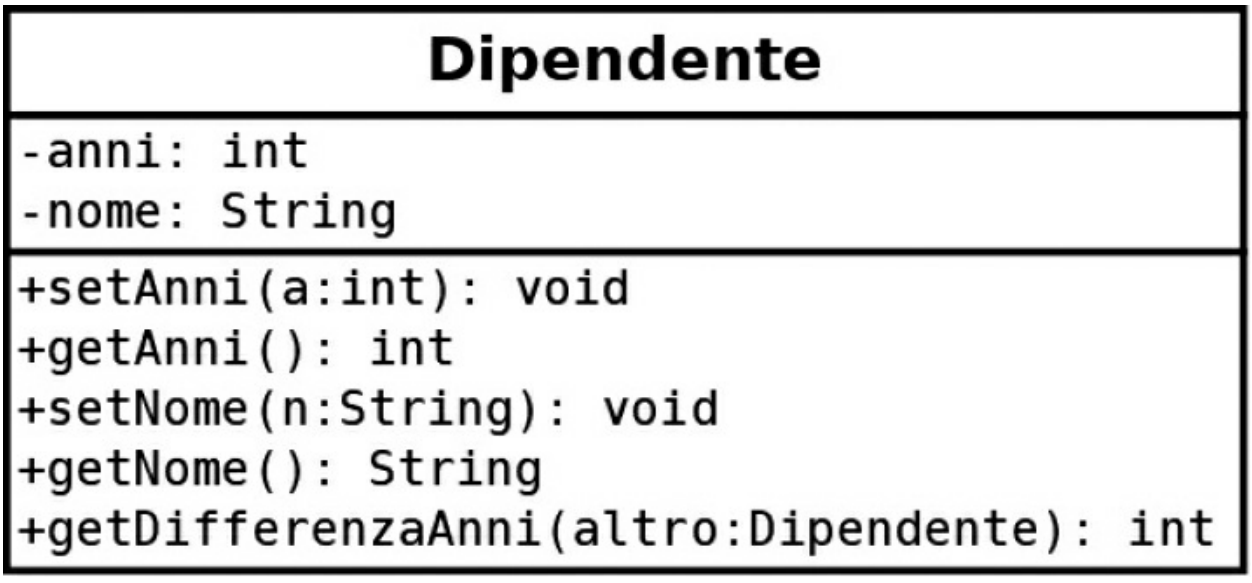


Figura 6.4 - Class Diagram “Classe Dipendente”.

L’implementazione in Java potrebbe essere la seguente:

```
public class Dipendente {
    private String nome;
    private int anni; //intendiamo età in anni
    //...
    public String getNome() {
        return nome;
    }
    public void setNome(String n) {
        nome = n;
    }
    public int getAnni() {

        return anni;
    }
    public void setAnni(int n) {
        anni = n;
    }
    public int getDifferenzaAnni(Dipendente altro) {
        return (anni – altro.anni);
    }
}
```

Nel metodo `getDifferenzaAnni()` notiamo che è possibile accedere direttamente alla variabile privata `anni` dell'oggetto `altro`, senza dover utilizzare il metodo `getAnni()`.

Il codice precedente è quindi valido per la compilazione, ma il seguente metodo:

```
    public int getDifferenzaAnni(Dipendente altro) {
        return (getAnni() – altro.getAnni());
    }
```

favorirebbe sicuramente di più il riuso di codice, e quindi è da considerarsi preferibile. Infatti, il metodo `getAnni()` si potrebbe evolvere introducendo controlli (magari per la sostituzione della variabile `anni` con la variabile `dataDiNascita`), che conviene richiamare piuttosto che riscrivere.

Il riuso dei metodi è molto importante. Se copiamo ed incolliamo il codice da un metodo ad un altro stiamo probabilmente facendo un errore. Infatti, come abbiamo detto più volte il software si evolve, ed avere due metodi che contengono lo stesso codice nel nostro programma, non ne favorirà l'evoluzione. Se per esempio scopriamo di avere un bug in uno di

questi due metodi, è facile dimenticarsi che c'è un altro metodo che contiene probabilmente lo stesso problema.

6.3.3 Il reference `this`

Il metodo `getDifferenzaAnni()` dell'esempio precedente potrebbe aver provocato nel lettore qualche dubbio. Sino ad ora avevamo dato per scontato che l'accedere ad una variabile d'istanza all'interno della classe dove è definita, fosse un "processo naturale" che non aveva bisogno di reference. Ad esempio, all'interno del metodo `getGiorno()` nella classe `Data` accedevamo direttamente alla variabile `giorno` senza utilizzare reference. Alla luce dell'ultimo esempio e considerando che potrebbero essere istanziati tanti oggetti dalla classe `Data`, ci potremmo chiedere: se `giorno` è una variabile d'istanza, a quale istanza appartiene? La risposta a questa domanda è: dipende dall'oggetto corrente, ovvero dall'oggetto su cui è chiamato il metodo `getGiorno()`. Per esempio, nella fase d'esecuzione di una certa applicazione, potrebbero essere istanziati due particolari oggetti, che supponiamo si chiamino `mioCompleanno` e `tuoCompleanno`. Entrambi questi oggetti hanno una propria variabile `giorno`. Ad un certo punto, all'interno del programma potrebbe presentarsi la seguente istruzione:

```
System.out.println(mioCompleanno.getGiorno());
```

Sarà stampato a video il valore della variabile `giorno` dell'oggetto `mioCompleanno`. Quindi il metodo `getGiorno()` che è definito dal seguente codice:

```
public int getGiorno() {
    return giorno;
}
```

deve usare un reference in qualche modo per la variabile `giorno`, perché nel momento in cui questo metodo viene chiamato sull'oggetto `mioCompleanno`, esso restituisce la variabile `giorno` dell'oggetto `mioCompleanno`, e non quello dell'oggetto `tuoCompleanno`. Ovvero è come se il metodo fosse definito così:

```
public int getGiorno() {
    return mioCompleanno.giorno;
}
```

Come è facile intuire non sarebbe possibile scrivere questo codice, visto che non sappiamo come si chiameranno gli oggetti che verranno istanziati da questa classe, e se avremo visibilità su di essi.

Anche in questo caso il compilatore prende iniziativa automaticamente. Se il programmatore non usa un reference per una certa variabile d'istanza, al momento della compilazione il codice sarà modificato dal compilatore stesso, che aggiungerà un reference all'oggetto corrente davanti alla variabile. Ma quale reference all'oggetto corrente? La classe non può conoscere a priori i reference degli oggetti che saranno istanziati da essa in fase di runtime!

Java introduce una parola chiave che per definizione coincide con il reference all'oggetto corrente: **`this`** (in italiano "questo"). Il reference `this` viene quindi implicitamente aggiunto nel bytecode compilato per referenziare ogni variabile d'istanza non esplicitamente referenziata.

Quindi il metodo `getGiorno()` che avrà a disposizione la JVM dopo la compilazione sarà modificato come segue:

```
    public int getGiorno() {
        return this.giorno; //il this lo aggiunge il
compilatore
    }
```

Anche in questo caso abbiamo notato un altro di quei comportamenti del linguaggio che una volta faceva definire Java "semplice" da imparare. Se sino a questo punto non ci siamo posti il problema dell'usare il reference `this` con le variabili d'istanza, è perché anche questa volta "Java ci ha dato una mano". Ripetiamo, oggi Java è un linguaggio complicato, ma almeno i suoi creatori hanno creato alcuni meccanismi che ne facilitano l'approccio.

6.3.4 Uso di `this` con variabili

Nel secondo capitolo abbiamo distinto le variabili d'istanza dalle variabili locali. La diversità tra i due concetti è tale che il compilatore ci consente di dichiarare nella stessa classe, una variabile locale (o un parametro di un metodo) ed una variabile d'istanza con lo stesso identificatore. Infatti la JVM

alloca le variabili locali e le variabili d'istanza in differenti aree di memoria, dette rispettivamente **Stack Memory** e **Heap Memory.** Nella memoria Stack vengono memorizzati temporaneamente i dati definiti da un metodo in esecuzione, e una volta terminato il metodo tali dati saranno rimossi dalla memoria). Nella Heap Memory vengono invece memorizzati gli oggetti e i propri dati, e questi saranno deallocati dalla Garbage Collection automaticamente quando questa si assicurerà che non siano più utilizzabili dal programma. La parola chiave `this` si inserisce in questo discorso nel seguente modo. Abbiamo più volte avuto a che fare con passaggi di parametri in metodi, al fine di inizializzare variabili d'istanza. Sino ad ora, per il parametro passato, siamo stati costretti ad inventare un identificatore differente da quello della variabile d'istanza da inizializzare. Consideriamo la seguente classe:

```
public class Cliente {
    private String nome, indirizzo, numeroDiTelefono;

    public void setDati (String n, String ind, String num) {
        nome = n;
        indirizzo = ind;
        numeroDiTelefono = num;
    }
    //...
}
```

Notiamo l'utilizzo dell'identificatore `n` per inizializzare `nome`, `num` per `numeroDiTelefono` e `ind` per `indirizzo`. Non c'è nulla di sbagliato in questo. Conoscendo però l'esistenza del reference `this`, abbiamo la possibilità di scrivere equivalentemente:

```
public class Cliente {
    private String nome, indirizzo, numeroDiTelefono;

    public void setDati (String nome, String indirizzo,
String numeroDiTelefono) {
        this.nome = nome;
        this.indirizzo = indirizzo;
        this.numeroDiTelefono = numeroDiTelefono;
    }
    //...
}
```

Infatti, tramite la parola chiave `this`, specifichiamo che la variabile referenziata appartiene all'istanza. Di conseguenza la variabile non referenziata sarà il parametro del metodo, senza che vi sia ambiguità.

Questo stile di programmazione è in generale considerato preferibile, anzi è da considerarsi uno standard. In questo modo, infatti, non c'è possibilità di confondere le variabili con nomi simili. Nel nostro esempio potrebbe capitare, per errore, di assegnare il parametro `n` alla variabile d'istanza `numeroDiTelefono` ed il parametro `num` alla variabile `nome`. Potremmo affermare che l'utilizzo di `this` aggiunge chiarezza al nostro codice.

Nella figura 6.5 è rappresentata la classe `Cliente` in un diagramma delle classi UML.

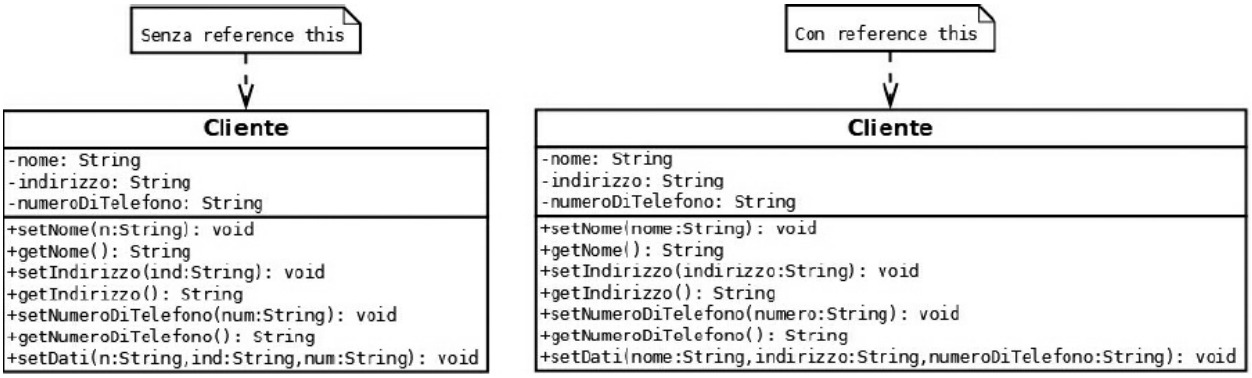


Figura 6.5 - Class Diagram "Classe Cliente".

Il lettore noti che se scrivessimo:

```
public class Cliente {
    private String nome, indirizzo, numeroDiTelefono;

    public void setNumeroDiTelefono(String numeroDiTelefono)
{
        numeroDiTelefono = numeroDiTelefono;
    }


    public void setIndirizzo(String indirizzo) {
        indirizzo = indirizzo;
    }

    public void setNome(String nome) {
        nome = nome;
```

```
    }
    //...
}
```

il compilatore, non trovando riferimenti espliciti, considererebbe le variabili sempre locali e quindi non otterremmo il risultato desiderato.

6.3.5 Uso di `this` con metodi e costruttori

Per quanto detto, `this` rappresenta l'oggetto corrente e può quindi referenziare le variabili d'istanza. Naturalmente con `this` è possibile anche invocare metodi. Per esempio:

```
public class Cliente {

    private String nome;
    private String indirizzo;
    private String numeroDiTelefono;

    public Cliente(String nome, String indirizzo, String
numeroDiTelefono) {
        this.setNome(nome);
        this.setIndirizzo(indirizzo);
        this.setNumeroDiTelefono( numeroDiTelefono);
    }

    public void setNumeroDiTelefono(String numeroDiTelefono)
{
        this.numeroDiTelefono = numeroDiTelefono;
    }

    public void setIndirizzo(String indirizzo) {
        this.indirizzo = indirizzo;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
    //...
}
```

Nel metodo costruttore abbiamo invocato i metodi setter sfruttando il reference `this`. In questo caso però, se non avessimo anteposto il reference

`this` ai metodi setter, avremmo comunque ottenuto lo stesso risultato, visto che `this` per Java rimane un reference implicito come visto anche per le variabili. Più interessante è l'uso che si fa solitamente del reference `this` quando abbiamo più costruttori nella stessa classe (overload di costruttori) e da un costruttore se ne vuole chiamare un altro **(riuso di costruttori).** La sintassi cambia leggermente visto che un costruttore non è un metodo qualsiasi e l'operatore dot non è utilizzabile. Facciamo un esempio:

```
public class Cliente {
    //...
    public Cliente (String nome, String indirizzo) {
        this(nome, indirizzo,"sconosciuto");
    }
    public Cliente(String nome, String indirizzo, String
numeroDiTelefono) {
        this.setNome(nome);
        this.setIndirizzo(indirizzo);
        this.setNumeroDiTelefono(numeroDiTelefono);
    }
    //...
}
```

Notare che abbiamo passato a `this` tre parametri di tipo stringa (non avendo ricevuto in input il numero di telefono abbiamo passato al secondo costruttore un valore di default) e quindi abbiamo chiamato il secondo costruttore (abbiamo riusato codice). In caso di overload di costruttori quindi, la raccomandazione è di far sì che questi si richiamino l'un l'altro per evitare di riscrivere più volte le stesse linee di codice. Nella figura 6.6 possiamo notare nella rappresentazione dell'ultima versione della classe `Cliente`, la sintassi dei costruttori nel diagramma delle classi. Viene utilizzata un cosiddetto stereotipo UML, ovvero una parola (di solito un sostantivo o un aggettivo) che caratterizza ciò che stiamo dichiarando. Nel nostro caso, con lo stereotipo `<<constructor>>` viene reso più evidente il fatto che stiamo dichiarando appunto un costruttore, e non un metodo qualsiasi.

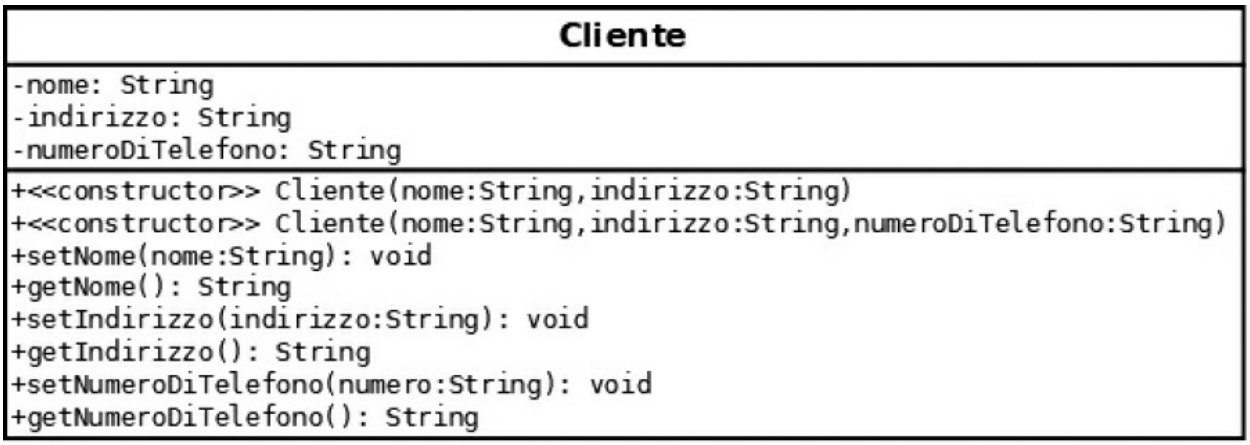


Figura 6.6 - Class Diagram "Classe Cliente con costruttori".

6.4 Quando utilizzare l'incapsulamento

Applicare l'incapsulamento non è mai sbagliato. Una qualsiasi classe di una qualsiasi applicazione potrebbe e dovrebbe essere sviluppata utilizzando l'incapsulamento. Non bisognerebbe mai permettere di sacrificare l'incapsulamento per risparmiare qualche secondo di programmazione. Qualche volta può sembrarci che usufruire dell'incapsulamento sia superfluo su determinate classi, ma l'esperienza ci insegna che incapsulare le nostre classi non è mai un errore. Facciamo un esempio banale. Abbiamo già accennato al fatto che per realizzare un'applicazione a qualsiasi livello (sempre che non sia veramente elementare), bisogna apportare modifiche incrementali. Il software si evolve, difficile che creato un programma non si modifichi più. Per esempio, supponiamo di voler scrivere una semplice applicazione che, assegnati due punti, disegni il segmento che li unisce. Supponiamo inoltre che si utilizzi la seguente classe non incapsulata `Punto`, già incontrata nel capitolo 2:

```
public class Punto {
    public int x, y;
    //...
}
```

l'applicazione, in un primo momento, istanzierà ed inizializzerà due punti con il seguente frammento di codice:

(**Codice 6.4**)

```
 Punto p1 = new Punto();
 Punto p2 = new Punto();
 p1.x = 5;
 p1.y = 6;
 p2.x = 10;
 p2.y = 20;
 //...
```

Supponiamo che l'evolversi della nostra applicazione renda necessario che i due punti non debbano trovarsi fuori da una certa area piana ben delimitata. Risulta quindi evidente che la soluzione migliore sia implementare l'incapsulamento all'interno della classe `Punto` in questo modo:

```
public class Punto {
    private int x, y;
    private final int VALORE_MASSIMO_PER_X=10;
    private final int VALORE_MINIMO_PER_X=-10;
    private final int VALORE_MASSIMO_PER_Y=10;
    private final int VALORE_MINIMO_PER_Y=-10;
    public void setX(int a) {
        if (a <= VALORE_MASSIMO_PER_X && a >=
VALORE_MINIMO_PER_X) {

        x = a;
        System.out.println("Valore per x valido: " + a);
    }
    else {
        System.out.println("valore per x non valido: " + a);
        }
    }
    public void setY(int a) {
        if (a <= VALORE_MASSIMO_PER_Y && a >=
VALORE_MINIMO_PER_Y) {
            y = a;
            System.out.println("Valore per y valido: " + a);
        }
        else {
            System.out.println("Valore per y non valido: " +
a);
        }
    }
    //...
}
```

Purtroppo però, dopo aver apportato queste modifiche alla classe `Punto` saremo costretti a modificare anche il frammento di codice 6.4 dell'applicazione nel modo seguente:

(**Codice 6.5**)

```
Punto p1 = new Punto();
Punto p2 = new Punto();
p1.setX(5);
p1.setY(6);
p2.setX(10);
p2.setY(20);
...
```

Saremmo partiti meglio se la classe `Punto` fosse stata incapsulata a priori:

```
public class Punto {
    private int x, y;
    public void setX(int a) {
        x = a;
    }
    public void setY(int a) {
        y = a;
    }
    ...
}
```

dal momento che avremmo modificato solo il codice all'interno dei metodi d'accesso e saremmo stati costretti ad utilizzare il codice 6.5 all'interno dell'applicazione che utilizza `Punto`.

Si noti che un codice simile al 6.5, potrebbe essere utilizzato in molte altre parti dell'applicazione. In un programma vero, composto da tante classi, se non si usasse l'incapsulamento ma si usasse un codice simile al 6.4, le modifiche da apportare (come per esempio aggiungere i controlli sulle coordinate del punto) sarebbero state numerose e dislocate in molte classi diverse.

Arriviamo alla conclusione che l'incapsulamento è sempre consigliabile in Java. Un linguaggio orientato agli oggetti "puro" come lo SmallTalk, infatti, non permetterebbe la dichiarazione di attributi pubblici. Java però vuole essere un linguaggio semplice da apprendere, e permettendo la dichiarazione di variabili d'istanza pubbliche, non costringe l'aspirante programmatore ad imparare prematuramente un concetto quale l'incapsulamento. In particolare, nei primi tempi, non se ne apprezzerebbe completamente l'utilizzo, dovendo approcciare a troppi concetti nuovi contemporaneamente.

6.5 Come usare l'incapsulamento

In generale, se parliamo solo del linguaggio e non della filosofia object oriented, l'incapsulamento si riduce semplicemente al creare classi che dichiarano private le proprie variabili e pubblici i relativi metodi set e get. Iniziamo quindi a prendere subito questa buona abitudine, le variabili delle nostre classi vanno sempre dichiarate private, fornendo a corredo i metodi setter e getter per renderle utilizzabili all'esterno della classe. Anche se fondamentale, è ovvio che questa attività possa risultare noiosa e soprattutto non è raro sbagliare a scrivere correttamente i metodi setter e getter. Basta una lettera minuscola di troppo e si finisce con il perdere tanto tempo. Fortunatamente ci vengono incontro i vari strumenti di sviluppo che consentono di automatizzare la creazione di questi metodi. Netbeans ed Eclipse posseggono menu contestuali che si aprono facendo clic con il tasto destro del mouse sull'editor, dove è presente la voce Refactoring (nel caso di Eclipse c'è anche un menu Source) che contiene varie voci per automatizzare parti di codice. La voce Encapsulate Field(s) genererà automaticamente i metodi setter e getter. Persino EJE permette di creare automaticamente questi metodi a partire dalla definizione della variabile da incapsulare. Basta aprire il menu inserisci e fare clic su Proprietà JavaBean (oppure premere CTRL-9). Per Proprietà JavaBean intendiamo una variabile d'istanza incapsulata. Un semplice wizard (procedura guidata) chiederà di inserire prima il tipo della variabile (per esempio "String") e poi il nome (per esempio "nome"). Una volta fornite queste due informazioni EJE adempirà al suo compito. Netbeans ha anche un altro menu contestuale veloce che si ottiene premendo contemporaneamente i tasti ALT + INSERT e che offre diverse voci per creare codice velocemente, tra le quali vi è Add Property che permette

proprio la creazione da zero di una variabile incapsulata.

> **Il nome “Proprietà JavaBean” deriva dal termine dei Java-Beans, una tecnologia che nei primi anni di Java ebbe molto successo, attualmente accantonata. Il nome JavaBean però è ancora oggi sulla bocca di tutti, anche grazie al riutilizzo di tale termine nella tecnologia EJB (Enterprise JavaBeans). In inglese vuol dire “chicco di Java” (ricordiamo che Java è il nome di una tipologia di caffè, cfr. appendice A on line).**

Ma bisogna per forza dichiarare entrambi i metodi set e get per ogni variabile? Di solito sì, ma se per esempio vogliamo limitare l’accesso della variabile d’istanza al di fuori della classe, è possibile anche evitare di dichiarare il metodo set (se non vogliamo che possa essere impostata al di fuori della classe), il metodo get (nel caso vogliamo impedirne la lettura al di fuori della classe), o anche entrambi i metodi (in questo caso la variabile esisterà e sarà gestita solo all’interno della stessa classe dove è stata definita). Ci sono per esempio casi in cui è necessario dichiarare classi le cui variabili d’istanza possono essere impostate una sola volta nel momento in cui si istanziano, e il cui valore non cambierà nel tempo. In quel caso basterà non fornire i metodi set, sfruttando un costruttore per il primo (e unico) assegnamento. Per esempio la seguente classe `Punto` astrae il concetto di punto fisso:

```
public class PuntoFisso {
    private int x, y;

    public PuntoFisso (int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
```

```
        return y;
    }
}
```

Il valore delle variabili `x` ed `y` non è modificabile una volta istanziato un oggetto, come nel modo seguente:

```
PuntoFisso puntoFisso = new PuntoFisso(3,5);
```

Questo perché non esistono i metodi set.

6.6 Gestione dei package

Il significato di `public` è chiaro quando applicato a variabili e metodi. Ma come mai sino ad ora abbiamo definito tutte le classi `public`? Per capirlo dobbiamo introdurre meglio il concetto di package. Abbiamo visto come la libreria standard di Java sia organizzata in package. Grazie a questo concetto, il programmatore ha quindi la possibilità di organizzare anche le proprie classi. È molto semplice infatti dichiarare una classe appartenente ad un package. La parola chiave **`package`** permette di specificare, prima della dichiarazione della classe, il package di appartenenza. Ecco un frammento di codice che ci mostra la sintassi da utilizzare:

```
package programmi.gestioneClienti;
public class AssistenzaClienti {
    . . . . .
```

> **L'istruzione `package` deve essere obbligatoriamente la prima in un file Java.**

In questo caso la classe `AssistenzaClienti` appartiene al package `gestioneClienti`, che a sua volta appartiene al package `programmi`. Dichiarare la classe `AssistenzaClienti` pubblica, permetterà a tutte le classi che vengono dichiarate anche esternamente al package `programmi.gestioneClienti` di utilizzarla. Se non avessimo anteposto la parola chiave `public` alla classe `AssistenzaClienti`, solo le classi appartenenti allo stesso package avrebbero potuto usare la classe

`AssistenzaClienti`.

I package fisicamente sono semplici cartelle (directory). Ciò significa che, dopo aver dichiarato la classe appartenente a questo package, dovremo inserire la classe compilata all'interno di una cartella chiamata `gestioneClienti`, situata a sua volta all'interno di una cartella chiamata `programmi`. Di solito il file sorgente va tenuto separato dalla classe compilata così come schematizzato in figura 6.7, dove abbiamo idealizzato i file come ovali e le directory come rettangoli.

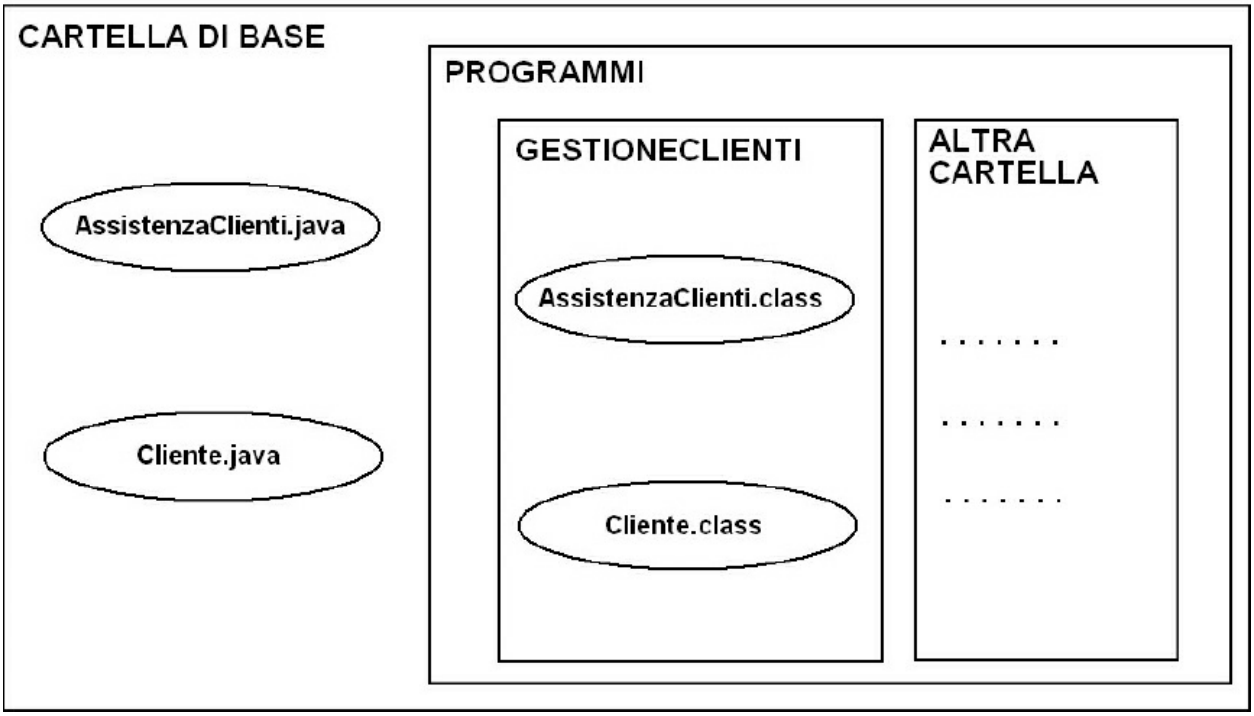


Figura 6.7 - Gestione dei package.

Il Java Development Kit ci consente comunque di realizzare il giusto inserimento dei file nelle cartelle e la relativa creazione automatica delle cartelle stesse, mediante il comando:

```
javac –d . AssistenzaClienti.java
```

Eseguendo il suddetto comando dalla cartella di base (dove è situato il file AssistenzaClienti.java), verranno automaticamente create le cartelle programmi e gestioneClienti, ed il file AssistenzaClienti.class collocato nella cartella giusta, senza però spostare il file sorgente dalla cartella di base. A questo punto, supponendo che il file in questione contenga il metodo `main()`,

potremmo eseguirlo solo posizionandoci nella cartella di base ed eseguendo il comando:

```
java programmi.gestioneClienti.AssistenzaClienti
```

infatti, a questo punto, il file è associato ad un package e non potrà essere più chiamato spostandosi nella cartella dove esso è presente utilizzando il comando:

```
java AssistenzaClienti
```

Inoltre bisogna utilizzare un comando di `import` per utilizzare tale classe da un package differente.

> **Esistono altri problemi collegati all'utilizzo dei package che l'aspirante programmatore potrà incontrare e risolvere con l'aiuto dei messaggi di errore ricevuti dal compilatore. Per esempio potrebbe risultare problematico utilizzare classi appartenenti ad un package, da una cartella non direttamente legata al package stesso. Ciò è risolvibile mediante l'impostazione della variabile di ambiente** classpath **del sistema operativo, che dovrebbe puntare alla cartella di base relativa al package in questione. All'argomento** classpath **è dedicata l'appendice E.**

6.6.1 Gestione manuale

Sebbene sia semplice da spiegare, la gestione dei package può risultare difficoltosa e soprattutto inutilmente noiosa se fatta a mano. Se il lettore è arrivato sino a questo punto utilizzando uno strumento come EJE o il Blocco Note per esercitarsi, vuol dire che è pronto per passare ad un editor più sofisticato. Un qualsiasi IDE Java gestisce l'organizzazione dei package in maniera automatica.

> **Attualmente EJE permette di specificare la directory di output dove devono essere generate le classi. Di default però**

EJE compila le classi nella stessa directory dove si trovano i sorgenti. Ci sono casi in cui per poter compilare correttamente i vostri file bisognerà compilare insieme tutti i file che compongono la vostra applicazione. Non si dovrebbero presentare problemi però se si specifica come directory di output una cartella diversa da quella di default (F12 - Opzioni). In generale tool complessi come Eclipse e Netbeans gestiscono in maniera trasparente i package.

Se il lettore invece volesse continuare ad utilizzare il Blocco Note a scopo didattico, ed iniziare ad utilizzare i package, è consigliabile seguire alla lettera le linee guida seguenti.

- Quando si inizia un progetto, è opportuno creare una cartella con il nome del progetto stesso. Per esempio se il progetto si chiama `Bancomat`, chiamare la cartella Bancomat.
- Poi vanno create almeno due sottocartelle, chiamate src (dove metteremo tutti i sorgenti) e classes (dove metteremo tutti i file compilati). In questo modo divideremo l'ambiente di sviluppo dall'ambiente di distribuzione. Opzionalmente consigliamo di creare altre cartelle parallele come docs (dove mettere tutta la documentazione), config (dove mettere i file di configurazione), etc. Per esempio, nella cartella Bancomat inserire le cartelle src, classes, etc.
- Ogni volta che una classe deve appartenere ad un package, creare la cartella-package a mano ed inserire il file sorgente all'interno. Per esempio, se la classe `ContoCorrente` deve appartenere al package `banca`, creare la cartella banca all'interno della cartella src e inserire il file ContoCorrente.java nella cartella banca.
- Ogni volta che dobbiamo compilare, bisogna sempre posizionarsi tramite la prompt di DOS nella cartella classes e da qui eseguire un comando del tipo:

```
javac –d ...\src\nomepackage\*.java
```

Se volessimo compilare tutte le classi del package `banca`, dovremmo eseguire il seguente comando:

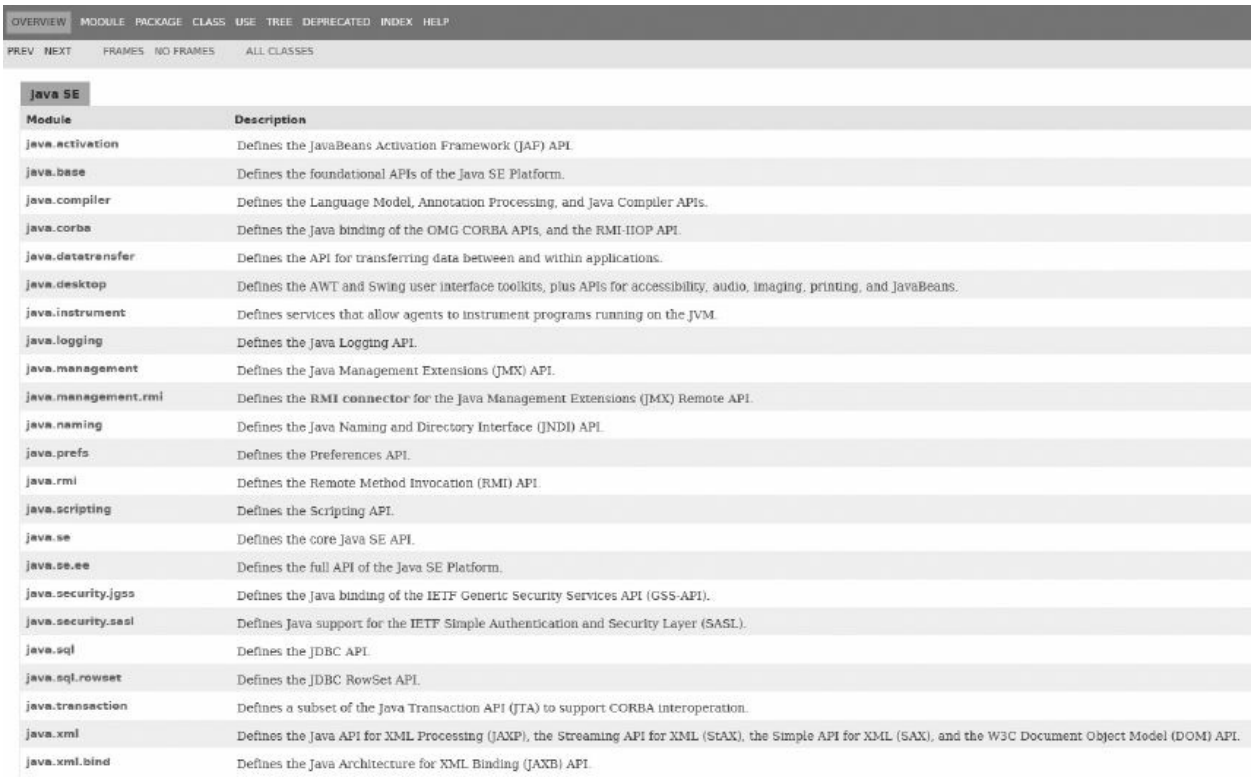
```
javac –d ...\src\banca\*.java
```

Consigliamo al lettore di crearsi file batch (.bat) per immagazzinare comandi così lunghi o ancora meglio utilizzare un tool come Ant (per informazioni consultare http://ant.apache.org).

> **Questo tipo di organizzazione è proprio quello implementato automaticamente dai tool più importanti come Eclipse e Netbeans.**

6.6.2 Moduli

La novità più importante di Java 9 è sicuramente l'introduzione dei **moduli.** Se i package rappresentano uno strumento per gestire la visibilità delle classi, i moduli invece rappresentano uno strumento più potente, che permette tra l'altro di esporre package con maggiore controllo, e di implementare quella che viene definita **strong encapsulation** (letteralmente **forte incapsulamento).** Con Java 9 anche la libreria standard è stata completamente riorganizzata utilizzando i package. Questo lo si può notare anche dalla documentazione ufficiale, come è possibile vedere nella pagina principale index.html, mostrata nella figura 6.8.



OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV NEXT FRAMES NO FRAMES ALL CLASSES

Java SE

Module	Description
java.activation	Defines the JavaBeans Activation Framework (JAF) API.
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.corba	Defines the Java binding of the OMG CORBA APIs, and the RMI-IIOP API.
java.datatransfer	Defines the API for transferring data between and within applications.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
java.instrument	Defines services that allow agents to instrument programs running on the JVM.
java.logging	Defines the Java Logging API.
java.management	Defines the Java Management Extensions (JMX) API.
java.management.rmi	Defines the RMI connector for the Java Management Extensions (JMX) Remote API.
java.naming	Defines the Java Naming and Directory Interface (JNDI) API.
java.prefs	Defines the Preferences API.
java.rmi	Defines the Remote Method Invocation (RMI) API.
java.scripting	Defines the Scripting API.
java.se	Defines the core Java SE API.
java.se.ee	Defines the full API of the Java SE Platform.
java.security.jgss	Defines the Java binding of the IETF Generic Security Services API (GSS-API).
java.security.sasl	Defines Java support for the IETF Simple Authentication and Security Layer (SASL).
java.sql	Defines the JDBC API.
java.sql.rowset	Defines the JDBC RowSet API.
java.transaction	Defines a subset of the Java Transaction API (JTA) to support CORBA interoperation.
java.xml	Defines the Java API for XML Processing (JAXP), the Streaming API for XML (StAX), the Simple API for XML (SAX), and the W3C Document Object Model (DOM) API.
java.xml.bind	Defines the Java Architecture for XML Binding (JAXB) API.

Figura 6.8 - Pagina principale della documentazione ufficiale.

Per scelta didattica, abbiamo deciso di dedicare all'argomento un capitolo intero del libro più avanti. Se non dichiariamo moduli infatti, possiamo utilizzare Java come veniva utilizzato fino a Java 8. Quindi per ora spiegheremo quali sono le regole di visibilità che valgono senza utilizzare i moduli. Quando affronteremo l'argomento sui moduli allora, torneremo su alcuni argomenti trattati da questo capitolo.

6.7 Modificatori d'accesso

Sino ad ora abbiamo visto che esistono `private` e `public`, due parole chiavi che rappresentano degli aggettivi con cui andiamo a definire delle caratteristiche per i membri della nostra classe. Questi aggettivi vengono detti **modificatori** e ne abbiamo già incontrati altri come `final` o `static`. Un modificatore è una parola chiave capace di cambiare il significato di un componente di un'applicazione Java. Come già asserito nel capitolo 1, un modificatore sta ad un componente di un'applicazione Java come un aggettivo sta ad un sostantivo nel linguaggio umano. Ogni modificatore si applica a uno o più tipologie di componenti Java. Tra queste tipologie sono

comprese classi, metodi, variabili d'istanza, variabili locali, attributi di metodi e costruttori, ma ne esistono anche altre di cui non abbiamo ancora parlato come le interfacce, le enumerazioni e le annotazioni.

I modificatori protagonisti di questo capitolo `public` e `private`, vengono meglio definiti come **modificatori di accesso** (o anche **specificatori d'accesso)** perché regolano essenzialmente la visibilità e l'accesso ad un componente Java. Esiste anche un terzo modificatore d'accesso chiamato `protected`, ed un'altra opzione è quella di **non usare modificatori d'accesso** (in questo caso si parla di **visibilità di package).**

6.7.1 Modificatore `public`

II modificatore **`public`** è stato usato più volte sinora. Sappiamo già che può essere applicato sia ad un membro (attributo o metodo) di una classe, sia ad una classe stessa. Sappiamo anche che un membro dichiarato pubblico sarà accessibile da una qualsiasi classe situata in qualsiasi package, e che per l'incapsulamento le variabili d'istanza non dovrebbero mai essere dichiarate pubbliche. Una classe dichiarata pubblica sarà anch'essa visibile da un qualsiasi package (a meno di dichiarazione di moduli).

Per esempio consideriamo la seguente classe pubblica che dichiara una variabile pubblica e un metodo pubblico:

```
package com.cdsc.test;

public class PublicInClasse {
    public int variabilePublic;

    public void metodoPublic() {
        System.out.println("Invocato metodo public");
    }
}
```

La classe è dichiarata pubblica e dichiara membri pubblici, quindi la seguente classe compilerà senza problemi:

```
package com.cdsc.test;

public class PublicInClasse {
    public int variabilePublic;
```

```
    public void metodoPublic() {
        System.out.println("Invocato metodo public");
    }
}
```

6.7.2 Modificatore `protected`

Anche se le sottoclassi saranno l'argomento principale del prossimo capitolo dedicato all'ereditarietà, per poter affrontare il discorso sul modificatore d'accesso **`protected`**, è utile quantomeno avere un'idea di cosa sia una sottoclasse. Una **sottoclasse** è una classe che estende un'altra classe ereditandone le caratteristiche. Per esempio, supponiamo di avere una classe `Dipendente` e una classe `Programmatore` che condividono alcune caratteristiche. In particolare supponiamo che entrambe le classi dichiarino alcune variabili `stipendio`, `matricola` e `annoDiAssunzione`, ma che la classe `Programmatore` dichiari anche un array `certificazioni`. Queste classi possono essere messe in relazione di **ereditarietà** tra loro, scrivendo semplicemente che la classe `Programmatore` **estende** la classe `Dipendente`. Così facendo, la classe `Programmatore` non deve ridichiarare tutte le caratteristiche in comune con la classe `Dipendente` (perché saranno ereditate). La classe `Dipendente` potrebbe essere codificata così:

```
public class Dipendente {
    private int matricola;
    private int annoDiAssunzione;

    public void setMatricola(int matricola) {
        this.matricola = matricola;
    }

    public int getMatricola() {
        return matricola;
    }

    public void setAnnoDiAssunzione(int annoDiAssunzione) {
        this.annoDiAssunzione = annoDiAssunzione;
    }

    public int getAnnoDiAssunzione() {
        return annoDiAssunzione;
    }
```

```
}
```

La classe `Programmatore` usa la parola chiave `extends` per ereditare i metodi pubblici della classe `Dipendente`, e li usa all'interno del costruttore:

```
public class Programmatore extends Dipendente {
    private String[] certificazioni;

    public Programmatore(int matricola, int
annoDiAssunzione) {
        setMatricola(matricola);
        setAnnoDiAssunzione(annoDiAssunzione);
    }

    public void setCertificazioni(String[] certificazioni) {
        this.certificazioni = certificazioni;
    }

    public String[] getCertificazioni() {
        return certificazioni;
    }
}
```

Il modificatore d'accesso `protected` definisce per un membro il grado più accessibile dopo quello definito da `public`. Un membro protetto sarà infatti accessibile all'interno dello stesso package ed in tutte le sottoclassi della classe in cui è definito, anche se non appartenenti allo stesso package.

Per esempio consideriamo la seguente classe che dichiara una variabile e un metodo protetti:

```
package com.cdsc.test;

public class ProtectedInClasse {
    protected int variabileProtected;

    protected void metodoProtected() {
        System.out.println("Invocato metodo protected");
    }
}
```

La seguente classe appartenente allo stesso package compilerà senza problemi:

```
package com.cdsc.test;

public class ProtectedClasseInPackage {
    public void metodoCheUsaProtected() {
        ProtectedInClasse protectedInClasse = new
ProtectedInClasse();
        protectedInClasse.metodoProtected();
        System.out.println(protectedInClasse.variabileProtected

    }
}
```

Invece, compilando la seguente classe appartenente ad un altro package:

```
package com.cdsc;

import com.cdsc.test.*;

public class ProtectedClasseFuoriPackage {
    public void metodoCheUsaProtected() {
        ProtectedInClasse protectedInClasse = new
ProtectedInClasse();
        protectedInClasse.metodoProtected();

        System.out.println(protectedInClasse.variabileProtected

    }
}
```

otterremo il seguente errore in compilazione:

```
ProtectedClasseFuoriPackage.java:8: error: metodoProtected()
has
  protected access in ProtectedInClasse
        protectedInClasse.metodoProtected();
                          ^
ProtectedClasseFuoriPackage.java:9: error: variabileProtected
has
  protected access in ProtectedInClasse
        System.out.println(protectedInClasse.variabileProtected)

                                             ^
2 errors
```

I metodi e le variabili dichiarati `protected` sono ereditati in tutte le sottoclassi, anche se quest'ultime appartengono ad altri package.

Tutto sommato, per quanto detto per l'incapsulamento, non è necessario usare questo modificatore per le variabili d'istanza per farle ereditare nelle sottoclassi.

È un errore comune dei neofiti utilizzare variabili d'istanza protette. Come si può notare nella classe `Programmatore` invece, si possono utilizzare le variabili private della superclasse utilizzando i metodi setter (ed eventualmente i getter) senza toccare l'incapsulamento. Approfondiremo questo discorso nel prossimo capitolo.

Un altro errore comune è quello di pensare che `protected` renda accessibili i membri di una classe alle sole sottoclassi, e non alle classi dello stesso package. Infine, diversamente dal modificatore `public`, il modificatore `protected` non è applicabile alla dichiarazione di una classe.

6.7.3 Nessun modificatore d'accesso

Possiamo **evitare di usare modificatori** sia relativamente ad un membro (attributo o metodo) di una classe, sia relativamente ad una classe stessa.

Se non anteponiamo modificatori d'accesso ad un membro di una classe, esso sarà accessibile solo da classi appartenenti al package dove è definito. C'è un'unica differenza tra membri dichiarati senza modificatore d'accesso e membri dichiarati con il modificatore `protected:` i membri protetti saranno ereditati come fossero pubblici anche nelle sottoclassi esterne al package di dichiarazione.

Se invece dichiariamo una classe appartenente ad un package senza anteporre alla sua definizione il modificatore `public`, la classe stessa sarà visibile solo dalle classi appartenenti allo stesso package. In questo caso si parla di **visibilità di package.** Per esempio consideriamo la seguente classe dichiarata senza modificatori:

```
package com.cdsc.test;
```

```
class ClasseSenzaModificatore {
}
```

Compilando la seguente classe che prova ad utilizzare `ClasseSenzaModificatore`:

```
public class TestClasseSenzaModificatore {
    public static void main(String args[]) {
        ClasseSenzaModificatore c = new
ClasseSenzaModificatore();
    }
}
```

otterremo il messaggio di errore:

```
TestClasseSenzaModificatore.java:6: error: cannot find symbol
        ClasseSenzaModificatore c = new
ClasseSenzaModificatore();
        ^
  symbol: class ClasseSenzaModificatore
  location: class TestClasseSenzaModificatore
TestClasseSenzaModificatore.java:6: error: cannot find symbol
        ClasseSenzaModificatore c = new
ClasseSenzaModificatore();
                                    ^
  symbol: class ClasseSenzaModificatore
  location: class TestClasseSenzaModificatore
2 errors
```

proprio perché `ClasseSenzaModificatore` non è visibile al di fuori del proprio package.

6.7.4 Modificatore `private`

Il modificatore **`private`** restringe la visibilità di un membro di una classe alla classe stessa. Come abbiamo visto nei precedenti paragrafi rappresenta la chiave dell'incapsulamento. Si usano spesso anche i metodi privati, che sono invocati da altri metodi della stessa classe. Per quanto riguarda le variabili d'istanza private, abbiamo già osservato precedentemente che esse se incapsulate possono comunque essere utilizzate nelle sottoclassi.

Infine, diversamente dal modificatore `public`, il modificatore `private`

non è applicabile alla dichiarazione di una classe.

Riassumendo quanto detto in questi paragrafi, possiamo far riferimento alla tabella seguente. Essa riporta i modificatori di accesso e la relativa visibilità (solo per i membri di una classe):

MODIFICATORE	STESSA CLASSE	STESSO PACKAGE	SOTTOCLASSE	OVUNQUE
`public`	SI	SI	SI	SI
`protected`	SI	SI	SI	NO
nessun modificatore	SI	SI	NO	NO
`private`	SI	NO	NO	NO

6.8 Il modificatore `static`

`static` è forse il più potente modificatore di Java... forse anche troppo! Con `static` la programmazione ad oggetti trova un punto di incontro con quella funzionale ed il suo uso deve essere quindi limitato a situazioni di reale e concreta utilità. In altre parole non bisogna intendere questo modificatore come una scorciatoia per evitare di creare oggetti. Infatti vedremo che i metodi statici sono molto simili alle funzioni della programmazione funzionale. Il programmatore che proviene da precedenti esperienze con questo tipo di programmazione potrebbe quindi essere tentato di abusare di questo concetto. Invece non dobbiamo dimenticare che Java è nato come linguaggio orientato agli oggetti, e sarebbe inutile studiarlo senza saperne sfruttarne le potenzialità.

Potremmo tradurre il termine `static` con “condiviso da tutte le istanze della classe”, oppure semplicemente con “della classe”. Quindi le regole dell’incapsulamento quando parliamo di questo modificatore sono alterate. È possibile incapsulare una variabile statica, ma quella variabile verrà comunque condivisa da tutti gli oggetti istanziati dalla stessa classe. In generale quindi l’incapsulamento non si applica a variabili statiche (che come vedremo tra poco dovrebbero essere usate solo in determinati casi).

Per quanto detto, un membro statico ha la caratteristica di poter essere utilizzato mediante una sintassi del tipo:

```
NomeClasse.nomeMembro
```

in luogo di:

```
nomeOggetto.nomeMembro
```

Possiamo usare **static** per marcare i metodi e gli attributi di una classe (quindi tutti i membri di una classe) e come vedremo tra poco anche i cosiddetti *inizializzatori*. Non è possibile invece applicare questo modificatore a variabili locali, parametri di metodi, costruttori o classi. Anche senza istanziare la classe, l'utilizzo di un membro statico provocherà il caricamento in memoria della classe contenente il membro in questione, che quindi, condividerà il ciclo di vita con quello della classe.

> **Si possono anteporre alla dichiarazione di un componente di un'applicazione Java anche più modificatori alla volta, senza tener conto dell'ordine in cui vengono anteposti. Una variabile dichiarata** `static` **e** `public` **per esempio, avrà quindi le stesse proprietà di una dichiarata** `public` **e** `static`.

6.8.1 Metodi statici

Un **metodo statico** è un metodo che è condiviso da tutti gli oggetti istanziati dalla classe in cui è dichiarato. Possiamo dire che un metodo statico appartiene alla classe stessa, e non ad un'istanza particolare.

Un esempio di metodo statico è il metodo `sqrt()` (che sta per "square root" ovvero "radice quadrata") della classe `Math` (package `java.lang`), che viene chiamato tramite la sintassi:

```
Math.sqrt(numero)
```

`Math` è quindi il nome della classe e non il nome di un'istanza di quella classe (lo si poteva intuire dal fatto che il nome inizia con lettera maiuscola). La ragione per cui la classe `Math` dichiara tutti i suoi metodi statici è facilmente comprensibile. Infatti, se istanziassimo due oggetti differenti dalla classe `Math`, `ogg1` e `ogg2`, i due comandi:

```
ogg1.sqrt(4);
```

e

```
ogg2.sqrt(4);
```

produrrebbero esattamente lo stesso risultato (2). Effettivamente non ha senso istanziare due oggetti di tipo “matematica”.

Un metodo dichiarato `static` e `public` può considerarsi una sorta di funzione (non a caso nella classe `Math` la funzione radice quadrata è stata resa pubblica e statica) e perciò questo tipo di approccio ai metodi dovrebbe essere limitato solo ai casi in cui l’esecuzione del metodo non dipenda dalle caratteristiche dell’oggetto su cui viene chiamato il metodo. Un metodo correttamente dichiarato `static` potrebbe essere uno che fa operazioni su una stringa passata in input. Per esempio il seguente metodo restituisce quante occorrenze ci sono all’interno della stringa `string`, del carattere `character`:

```
public class StringUtils {
    public static int contaOccorrenze(String string, char
character) {
        int count = 0;
        for (int i = 0; i < string.length(); i++) {
            if (string.charAt(i) == character) {
                count++;
            }
        }
        return count;
    }
//Potremmo aggiungere altri metodi che fanno operazioni
sulle stringhe
}
```

In questo caso dichiarare il metodo `static` è di sicuro una buona scelta. In JShell possiamo per esempio eseguire questo comando (ovviamente dopo aver dichiarato anche la classe `StringUtils`):

```
jshell> StringUtils.contaOccorrenze("E il naufragar m\'è dolce
in
  questo mare",'a')
$3 ==> 4
```

Per invocare il metodo `contaOccorrenze()` abbiamo utilizzato il nome della

classe (è inutile istanziarla). Abbiamo passato una stringa come primo parametro e come secondo parametro un carattere, e come si può vedere dall'output, sono state trovare quattro occorrenze.

6.8.2 Variabili statiche (di classe)

Una **variabile statica**, essendo condivisa da tutte le istanze della classe, assumerà lo stesso valore per ogni oggetto di una classe.

Di seguito viene presentato un esempio:

```
public class ClasseDiEsempio {
    public static int a = 0;
}

public class ClasseDiEsempioPrincipale {
    public static void main (String args[]) {
        System.out.println("a = " + ClasseDiEsempio.a);
        ClasseDiEsempio ogg1 = new ClasseDiEsempio();
        ClasseDiEsempio ogg2 = new ClasseDiEsempio();

        ogg1.a = 10;
        System.out.println("ogg1.a = " + ogg1.a);
        System.out.println("ogg2.a = " + ogg2.a);
        ogg2.a = 20;
        System.out.println("ogg1.a = " + ogg1.a);
        System.out.println("ogg2.a = " + ogg2.a);
    }
}
```

L'output di questo semplice programma sarà:

```
a = 0
ogg1.a = 10
ogg2.a = 10
ogg1.a = 20
ogg2.a = 20
```

Come si può notare, se un'istanza modifica la variabile statica, essa risulterà modificata anche relativamente all'altra istanza. Infatti essa è condivisa dalle due istanze e risiede proprio nella classe. Una variabile di questo tipo potrebbe ad esempio essere utile per contare il numero di oggetti istanziati da una classe (per esempio incrementandola in un costruttore). Per esempio:

```
public class Counter {
    private static int counter = 0;
    private int number;
    public Counter() {
        counter++;
        setNumber(counter);
    }
    public void setNumber(int number) {
        this.number = number;
    }
    public int getNumber() {
        return number;
    }
}
```

Si noti come la variabile d'istanza `number` sia valorizzata nel costruttore di questa classe con il valore della variabile statica `counter`.

> **A differenza di una variabile d'istanza ordinaria, una variabile statica non viene inizializzata al valore nullo del suo tipo al momento dell'istanza dell'oggetto: non è una variabile d'istanza.**

Questo significa che `number` rappresenterà proprio il numero seriale di ogni oggetto istanziato. Per esempio, dopo questa istruzione:

```
Counter c1 = new Counter();
```

La variabile statica `counter` varrà `1` e la variabile `c1.number` varrà sempre `1`. Se poi istanziamo un altro oggetto `Counter`:

```
Counter c2 = new Counter();
```

allora la variabile statica `counter` varrà `2`. Infatti, essendo condivisa da tutte le istanze della classe, non viene azzerata nuovamente per ogni istanza. Invece la variabile `c1.number` varrà sempre `1`, mentre la variabile `c2.number` varrà `2`.

Il modificatore `static` quindi prescinde dal concetto di oggetto e lega strettamente le variabili al concetto di classe, che a sua volta si innalza a

qualcosa più di un semplice mezzo per definire oggetti. Per questo motivo a volte ci si riferisce alle variabili d'istanza statiche come **variabili di classe.**

> **Una variabile dichiarata `static` e `public` può considerarsi una sorta di "variabile globale". Le variabili globali rappresentano una scorciatoia per i programmatori, ma sono difficili da gestire e per questo sono a ragione considerate esempio di cattiva progettazione. Questo tipo di approccio alle variabili quindi va evitato. È però utile a volte utilizzare costanti globali, definite con `public, static` e `final`. Per esempio la classe `Math` definisce due costanti statiche pubbliche: `PI` ed `E` (cfr. documentazione ufficiale).**

Un metodo statico non può utilizzare variabili d'istanza senza referenziarle, ma solo variabili statiche (ed ovviamente variabili locali). Infatti, un metodo statico non appartiene a nessuna istanza in particolare, e quindi non potrebbe "scegliere" una variabile d'istanza di una istanza particolare senza referenziarla esplicitamente. Per esempio la seguente classe:

```
public class StaticMethod {
    private int variabileDiIstanza;
    private static int variabileDiClasse;
    public static void main(String args[]) {
        System.out.println(variabileDiIstanza);
    }
}
```

produrrà il seguente errore in compilazione:

```
non-static variable variabi
leDiIstanza cannot be referenced from a
static context
       System.out.println(variabileDiIstanza);
                          ^
1 error
```

Se invece stampassimo la variabile `variabileDiClasse` il problema non si porrebbe. Avevamo già inconsapevolmente incontrato una variabile statica,

più e più volte: la variabile `System.out`. Infatti `out` è una variabile statica della classe `System` (appartenente al package `java.lang`). Si tratta di una variabile di tipo `PrintStream`, che è un tipo di classe che permette di stampare (print) un flusso (stream) di dati su una certa destinazione, ed è questa classe che dichiara i metodi `print()` e `println()` che abbiamo utilizzato sinora (cfr. documentazione ufficiale). In particolare l'oggetto `System.out`, rappresenta l'**output standard del sistema**, ovvero, nel caso di Windows, rappresenta la prompt dei comandi. Ecco perché una chiamata a `System.out.println()` stampa il risultato sulla prompt dei comandi.

6.8.3 Inizializzatori statici ed inizializzatori d'istanza

Il modificatore `static` può anche essere utilizzato per marcare un semplice blocco di codice definito all'interno di una classe (ma al di fuori di un metodo). Stiamo parlando di un nuovo componente (per la verità utilizzato raramente) che viene chiamato **inizializzatore statico.** Un blocco statico definito all'interno di una classe avrà la caratteristica di essere chiamato al momento del caricamento in memoria della classe stessa, addirittura prima di un eventuale costruttore. La sintassi è semplice come mostrato nel seguente esempio:

```
public class EsempioStatico {
    private static int a = 10;
    public EsempioStatico() {
        a += 10;
    }
    static {
        System.out.println("valore statico = " + a);
    }
}
```

Anche questo blocco, come nel caso dei metodi statici, potrà utilizzare variabili definite fuori da esso se e solo se dichiarate statiche.

Supponiamo di istanziare un oggetto di questa classe mediante la seguente sintassi:

```
EsempioStatico ogg = new EsempioStatico();
```

Questo frammento di codice produrrà il seguente output:

```
valore statico = 10
```

Infatti, quando si istanzia un oggetto da una classe, questa deve essere prima caricata in memoria. È in questa fase di caricamento che viene eseguito il blocco statico, e di conseguenza stampato il messaggio di output. Successivamente viene chiamato il costruttore che incrementerà il valore statico.

È possibile inserire in una classe anche più inizializzatori statici; questi verranno eseguiti in maniera sequenziale dall'alto in basso.

L'uso di un inizializzatore statico può effettivamente essere considerato sporadico e legato a soluzioni di progettazione avanzate.

Esiste anche un'altra tipologia di inizializzatore, ma non statico. Si chiama **inizializzatore d'istanza (instance initializer** o **object initializer)** e si implementa includendo codice in un blocco di parentesi graffe all'interno di una classe. La sua caratteristica è quella di essere eseguito quando viene istanziato un oggetto, *prima* del costruttore. Per esempio, se istanziassimo la seguente classe:

```
public class InstanceInitializer {
    public InstanceInitializer() {
        System.out.println("Costruttore");
    }
    {
        System.out.println("Inizializzatore");
    }
}
```

l'output risultante sarebbe il seguente:

```
Inizializzatore
Costruttore
```

Anche per l'inizializzatore d'istanza, le situazioni in cui è necessario utilizzarlo sono molto rare.

> **È possibile inserire nella stessa classe più inizializzatori statici e d'istanza. Gli inizializzatori saranno eseguiti in maniera sequenziale a seconda di come sono stati ordinati all'interno del file, ma quelli statici avranno sempre la precedenza perché saranno eseguiti quando viene caricata la classe.**

6.8.4 Static import

Nella versione 5 di Java, furono introdotti (insieme a numerose altre novità) anche i cosiddetti **import statici.** Con il comando `import` siamo soliti importare nei nostri file classi ed interfacce direttamente da package esterni. Ovvero è possibile importare all'interno dei nostri file i nomi delle classi e delle interfacce, in modo tale che si possano poi usare senza dover specificare l'intero **fully qualified name** (nome completo di package). Per esempio, volendo utilizzare la classe `DOMSource` del package `javax.xml.transform.dom`, abbiamo due possibilità:

1. Importiamo la classe con un'istruzione come la seguente:

```
import javax.xml.transform.dom.DOMSource;
```

   per poi sfruttarla nel nostro file sorgente utilizzandone solo il nome. Per esempio:

```
DOMSource source = new DOMSource();
```

2. Oppure possiamo scrivere il nome completo di package ogni volta che la utilizziamo. Per esempio:

```
javax.xml.transform.dom.DOMSource source =
    new javax.xml.transform.dom.DOMSource();
```

Come è possibile immaginare la prima soluzione è preferibile.

In alcuni casi, però, potremmo desiderare di importare nel file solo ciò

che è dichiarato statico all'interno di una certa classe, e non la classe stessa. Per esempio, sapendo che la classe `Math` contiene solo metodi e costanti statiche, potremmo voler importare, piuttosto che la classe `Math`, solo i suoi membri statici. Per fare questo, bisogna utilizzare il seguente comando di import statico:

```
import static java.lang.Math.*;
```

In questo caso abbiamo importato tutti i membri statici all'interno del file. Quindi sarà lecito scrivere, in luogo di:

```
double d = Math.sqrt(4);
```

direttamente:

```
double d = sqrt(4);
```

senza anteporre il nome della classe al metodo statico. È anche possibile (e preferibile) importare staticamente solo alcuni membri specifici, per esempio:

```
import static java.lang.Math.PI;
import static java.lang.Math.random;
import static java.sql.DriverManager.getConnection;
import static java.lang.System.out;
```

Si noti come l'`import` di nomi di metodi statici non specifichi le parentesi con i relativi argomenti.

L'utilizzo di import statici non è sempre vantaggioso, ed in generale ne sconsigliamo l'utilizzo.

> **Tutte le considerazioni e i consigli di utilizzo di questa caratteristica di Java sono trattati nell'appendice on line F dal titolo "Approfondimento sugli import statici".**

6.8.5 Quando usare `static`

Il lettore avrà sicuramente notato che l'uso di `static` è stato scoraggiato praticamente in ogni paragrafo. In effetti se volessimo atteggiarci a puristi dell'Object Orientation, dovremmo affermare che `static` è semplicemente

una parola da non usare nella programmazione Java. Se ne potrebbe fare tranquillamente a meno. Ma `static` non è l'impersonificazione del male! Sicuramente bisogna evitare variabili globali, non creare funzioni al posto di metodi, abusare di import statici senza motivo. Ci sono casi in cui si possono risolvere problemi di programmazione in maniera elegante e funzionale. Per esempio nel caso del contatore di istanze visto nel paragrafo 6.8.2, una variabile statica ci ha consentito di realizzare il nostro obiettivo con il minimo sforzo. Le costanti statiche (attributi dichiarati `public, final` e `static)` poi, sono usatissime. Abbiamo anche visto che il metodo statico `sqrt()` della classe `Math`, è correttamente definito (perché istanziare un oggetto della classe `Math` senza che sia necessario?). Per quanto riguarda l'incapsulamento (argomento principale di questo capitolo) è possibile creare attributi statici e incapsularli con i metodi setter e getter (sempre statici), ma dobbiamo tenere presente che c'è una differenza. Consideriamo il seguente codice:

```
public class IncapsulamentoStatico {
    private static int attributoStatico = 0;
    public static void setAttributoStatico(int
attributoStatico) {
        this.attributoStatico = attributoStatico;
    }
    public static int getAttributoStatico() {
        return attributoStatico;
    }
}
```

Il codice non compilerà! Di seguito l'output della compilazione:

```
IncapsulamentoStatico.java:4: error: non-static variable this
cannot
  be referenced from a static context
       this.attributoStatico = attributoStatico;
       ^
IncapsulamentoStatico.java:4: warning: [static] static
variable should
  be qualified by type name, IncapsulamentoStatico, instead of
by an
  expression
       this.attributoStatico = attributoStatico;
           ^
1 error
```

```
1 warning
```

Questo perché il reference `this` abbiamo visto che si riferisce all'oggetto corrente e quindi non può essere usato nel contesto di un metodo statico, e questo viene segnalato come errore. Inoltre viene segnalato un warning (non è un errore ma solo un avvertimento del compilatore) che ci avverte che `this` si riferisce ad un oggetto, ma la variabile `attributoStatico` appartiene alla classe `IncapsulamentoStatico`, e sarebbe preferibile usare la classe per referenziare l'attributo.

Il codice quindi compilerà se il metodo setter verrà modificato nel seguente modo:

```
    public static void setAttributoStatico(int
attributoStatico) {
        IncapsulamentoStatico.attributoStatico =
attributoStatico;
    }
```

Di fatto quindi è possibile sfruttare l'incapsulamento per variabili statiche, tenendo presente che così non incapsuleremo oggetti, bensì classi. Un'implementazione dell'incapsulamento statico molto famosa (e molto utile) è il design pattern Singleton.

6.8.6 Design Pattern Singleton

I **Design Pattern** sono soluzioni progettuali standard applicabili a contesti diversi. Rappresentano un esempio di alto livello di riuso, non di un metodo o di una classe, ma di una soluzione progettuale (riuso concettuale). Se durante la fase di progettazione di un'applicazione si presenta un problema progettuale, la maggior parte delle volte non c'è bisogno di reinventare l'acqua calda. Una soluzione standard esiste già (è stata trovata e formalizzata in passato da progettisti esperti) ed è applicabile, basta solo conoscerla ed implementarla nel nostro contesto (ma non è sempre semplice!).

On line troverete l'appendice D dedicata all'introduzione dei Design Pattern.

Un ottimo esempio di Design Pattern è il **Singleton.** Se abbiamo la necessità che una classe debba essere istanziata una sola volta, e che quindi tutti gli utilizzatori di questa classe utilizzino sempre la stessa istanza, la soluzione consiste nell'utilizzo del pattern Singleton. L'implementazione classica del pattern Singleton, consiste nel creare una classe con:

1. un costruttore privato. In questo modo il costruttore che è necessario chiamare per istanziare un oggetto da questa classe è utilizzabile solo all'interno della classe stessa;
2. una variabile privata e statica dello stesso tipo della classe (di solito chiamata `instance` ovvero "istanza");
3. un metodo statico pubblico (solitamente chiamato `getInstance()`) che definisce una semplice logica per restituire sempre la stessa istanza (unica) della classe stessa.

Ecco un esempio di implementazione del pattern Singleton:

```
public class SingletonExample {
    private static SingletonExample instance;

    private SingletonExample() {
    }

    public static SingletonExample getInstance() {
        if (instance == null) {
            instance = new SingletonExample();
        }
        return instance;
    }
}
```

Notare come il metodo `getInstance()` essendo interno alla classe possa utilizzare il metodo costruttore (anche se privato) per istanziare la classe stessa. Per la logica che definisce sarà creato un oggetto solo la prima volta che sarà chiamato questo metodo ed assegnato all'attributo statico `instance`. Dalla seconda chiamata in poi questo metodo restituirà sempre la stessa istanza.

Quindi per ottenere l'unica istanza della classe `SingletonExample`, le

altre classi dovranno usare questa sintassi:

```
SingletonExample unicaIstanza =
SingletonExample.getInstance();
```

e saremo sicuri che tutte queste classi utilizzeranno sempre la stessa istanza di `SingletonExample`. Nella figura 6.9 si può osservare il diagramma delle classi che mostra l'implementazione del pattern Singleton appena descritto.

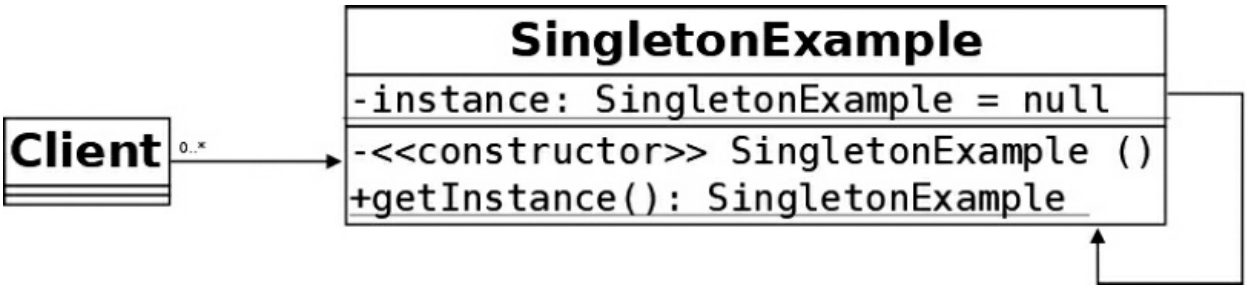


Figura 6.9 - Class Diagram "Singleton Pattern".

Nel diagramma di figura 6.9 possiamo notare due nuovi elementi di sintassi UML. Variabili e metodi statici vengono sottolineati in UML. Invece le due frecce (linee con **direzione)** presenti sul diagramma vengono dette **associazioni.** Le associazioni sono relazioni tra classi, ma più precisamente, rappresentano relazioni tra oggetti. Può sembrare strano perché il class diagram è un diagramma statico ma definisce elementi (le associazioni) che rappresentano relazioni dinamiche tra oggetti. L'informazione che ci danno le associazioni riguardano l'uso. Nel diagramma la freccia che parte dalla classe `Client` e termina nella classe `SingletonExample`, sta ad indicare che gli oggetti della classe `Client` al runtime, utilizzeranno un oggetto della classe `SingletonExample`. Il simbolo `0..*` che si può osservare alla fonte della associazione accanto alla classe `Client`, indica una delle **molteplicità dell'associazione.** Questa informazione rende evidente che l'associazione si riferisce ad oggetti, visto che la classe è sempre una e quindi non ha senso parlare di più classi, semmai di più oggetti. All'altra estremità dell'associazione non è indicata la molteplicità, il che indica che dobbiamo assumere che la molteplicità degli oggetti della classe `SingletonExample` assume il valore di default, ovvero `1`. Quindi, tenendo presente la direzione e le molteplicità, e il contenuto che definisce la classe `SingletonExample` come un Singleton, quest'associazione deve essere letta come "al runtime più

oggetti di tipo `Client` utilizzano un solo oggetto di tipo `SingletonExample`". Il che significa che ogni oggetto `Client` dichiara al suo interno in qualche modo un reference dell'oggetto `SingletonExample` e lo utilizza. L'altra associazione che vediamo in figura 6.9 invece parte e termina nella stessa classe, senza specificare le molteplicità. Si dice **associazione riflessiva**, ed implica che la classe `SingletonExample` possiede un riferimento verso sé stessa. E infatti `SingletonExample` è stata implementata proprio con un reference di sé stessa al suo interno.

Si noti che molteplicità e direzione delle associazioni sono facoltative, perché non è detto che in un certo momento dello sviluppo ci sia la sicurezza di avere certe informazioni, e quindi di poter prendere certe decisioni.

L'utilizzo dei pattern permette anche ai programmatori di usare una terminologia semplificata per parlare del codice. I programmatori spesso comunicano utilizzando la terminologia dei pattern (per esempio "... questa classe è un Singleton..."). Inoltre alcuni IDE automatizzano le implementazioni dei design pattern in maniera automatica.

Anche EJE permette di implementare automaticamente il pattern Singleton. Dopo aver creato una classe, semplicemente fare clic sul menu Insert⇨Singleton o usare la combinazione di tasti CTRL-F9.

Riepilogo

In questo capitolo è stato introdotto il supporto che offre Java all'**Object Orientation.** Dopo una panoramica storica sono stati elencati alcuni fondamentali paradigmi, ovvero astrazione, riuso, incapsulamento, ereditarietà e polimorfismo. Sono stati formalizzati il concetto di astrazione e quello di riuso. L'**astrazione** consiste nel creare le entità di programmazione (classi, metodi, etc.) con un nome e un'implementazione appropriati. Possiamo concentrarci sugli elementi che maggiormente caratterizzano l'entità che stiamo definendo e possiamo così rendere più agevole l'interazione col software. Il **riuso** ci consente di riutilizzare classi, metodi, package, soluzioni progettuali, concetti e altro, senza dover per forza

riscriverli. Anche essendo paradigmi fondamentali della programmazione orientata agli oggetti, l'astrazione e il riuso sono paradigmi validi anche per la programmazione funzionale. Per questa ragione sono spesso considerati paradigmi secondari, ma il lettore dovrebbe tenere ben presente che il loro utilizzo è assolutamente cruciale per programmare in Java, come cercheremo di dimostrare in questo manuale.

Il **diagramma delle classi** ci permette di rappresentare le classi in maniera molto leggibile e pratica. Con la notazione di associazione riusciamo ad ottenere anche informazioni sulle relazioni tra gli oggetti istanziati dalle classi. L'**incapsulamento** garantisce maggiore manutenibilità, robustezza e riusabilità. Scrivere codice per implementare incapsulamento è molto semplice (variabile privata e metodi accessor e mutator), tanto che la maggior parte degli strumenti di sviluppo supporta un modo standard per incapsulare gli attributi di una classe. Inoltre è anche semplice capire quando applicare questo concetto: praticamente sempre! Abbiamo anche introdotto il reference `this` anche se per ora non sembra una parola chiave indispensabile per programmare. La sua introduzione però, è stata utile per chiarire alcuni punti oscuri che potevano (anche inconsciamente) rappresentare un ostacolo alla completa comprensione degli argomenti. Infatti con `this` e l'operatore dot possiamo referenziare metodi e variabili presenti nella stessa classe, e con la sintassi `this()`, possiamo invocare un costruttore da un altro per risfruttarne il codice (riuso). Sono stati definiti anche il concetto di **package** e l'eventuale gestione "a mano" che potremmo usare se lavorassimo da riga di comando senza un IDE. Nonostante nella programmazione reale i package siano utilizzati sempre, abbiamo sottolineato quanto questo possa inizialmente influire sulla curva d'apprendimento di un lettore che si sta solo avvicinando alla programmazione Java, e vuole usare tutti i costrutti. Tuttavia, usando un IDE come Eclipse o Netbeans, l'utilizzo di package è praticamente trasparente.

I **moduli** saranno spiegati in un capitolo a parte più avanti, quando i tempi saranno maturi per apprezzare meglio i concetti. Intanto possiamo notare come tutta la libreria standard sia stata riorganizzata con i moduli in Java 9.

Sono stati poi spiegati i significati dei **modificatori d'accesso** e la loro applicabilità ai componenti del linguaggio Java. Il modificatore **`public`** è

possibile applicarlo a metodi, variabili e classi, che poi ereditano la proprietà di poter essere utilizzati da qualsiasi classe. Il modificatore **`protected`** invece rende un metodo o una variabile accessibile alle classi presenti nello stesso package, e a tutte le sottoclassi. Se non viene fornito **alcun modificatore** a una variabile o ad un metodo, allora questi saranno accessibili solo alle classi presenti nello stesso package. Infine il modificatore **`private`** rende un metodo o una variabile non accessibile all'esterno.

Inoltre è stato spiegato un modificatore fondamentale, **`static`**, e la sua applicabilità a metodi e variabili (che vengono chiamate variabili di classe). Il modificatore `static`, fa diventare metodi e variabili appartenenti alla classe, e non ad uno specifico oggetto. Per tale ragione bisogna referenziarli attraverso il nome della classe e non quello dell'oggetto.

Ne abbiamo approfittato per introdurre anche un nuovo tipo di componente: gli inizializzatori (statici e di oggetto). Un **inizializzatore statico** è composto dal modificatore `static` e un blocco di codice seguente. Ha la caratteristica di essere chiamato una sola volta nel momento in cui si carica la classe in memoria. Un **inizializzatore di oggetto** invece viene rappresentato semplicemente con un blocco di codice, che sarà chiamato prima del costruttore al momento in cui si istanzia un oggetto.

Abbiamo mostrato anche l'esistenza degli **import statici** e ne abbiamo per ora sconsigliato l'utilizzo. Con essi riusciamo a importare in una classe solo i membri statici di altre classi. Inoltre abbiamo dimostrato come questo modificatore sia adatto a risolvere problemi particolari, e che non è uno strumento di cui abusare. Agli import statici è dedicato un approfondimento nell'appendice on line F.

Ne abbiamo approfittato per introdurre i **Design Pattern** e in particolare il pattern **Singleton.** I Design Pattern ci permettono di utilizzare soluzioni progettuali già consolidate per risolvere alcuni problemi tipici della programmazione. Il pattern Singleton in particolare, consente di creare una classe da cui si può istanziare un unico oggetto. In questo modo si è sicuri che tutti gli oggetti istanziati da altre classi dell'applicazione avranno a che fare sempre con questo unico oggetto.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere le ragioni della nascita della programmazione ad oggetti (unità 6.1)	☐	
Saper elencare i paradigmi ed i concetti fondamentali della programmazione ad oggetti (unità 6.2)	☐	
Saper creare un diagramma delle classi di base (unità 6.2, 6.9)	☐	
Saper definire ed utilizzare il concetto di astrazione e di riuso (unità 6.2)	☐	
Comprendere l'utilizzo e l'utilità dell'incapsulamento (unità 6.3, 6.4, 6.5)	☐	
Comprendere l'utilizzo e l'utilità del reference `this` (unità 6.3)	☐	
Capire quando e come usare l'incapsulamento (unità 6.4, 6.5)	☐	
Comprendere l'utilizzo e l'utilità dei package (unità 6.6, 6.7)	☐	
Capire e saper utilizzare i modificatori d'accesso (unità 6.7)	☐	
Capire e saper utilizzare il modificatore `static` (unità 6.8)	☐	
Saper accennare al concetto di Design Pattern e comprendere l'implementazione del pattern Singleton (unità 6.8)	☐	

Note:

7

Ereditarietà ed interfacce

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Comprendere la definizione, l'utilizzo e l'utilità dell'ereditarietà (generalizzazione e specializzazione) (unità 7.1, 7.5).
- ✔ Conoscere le conseguenze dell'utilizzo contemporaneo di incapsulamento ed ereditarietà (unità 7.4).
- ✔ Comprendere e saper utilizzare i modificatori `abstract` e `final` (unità 7.3, 7.6).
- ✔ Comprendere e saper utilizzare la parola chiave `super` (unità 7.5).
- ✔ Comprendere e saper utilizzare le interfacce (unità 7.7).
- ✔ Sapersi destreggiare nei vari casi complessi dell'ereditarietà multipla (unità 7.7).

In questo capitolo affronteremo l'**ereditarietà** ed altri concetti ad essa correlati. Qui non parleremo solo ed esclusivamente di programmazione ma anche di concetti più astratti; dovremo imparare ad avere una visione da progettisti. Chiunque può programmare, basta conoscere come si usa l'`if` e il `for`, e si può fare tutto! In teoria è così, in pratica bisogna avere delle regole e le idee chiare prima di mettersi a programmare a testa bassa. Quello che fa la

differenza tra un bravo smanettone e un bravo sviluppatore è la qualità del suo lavoro. Uno sviluppatore dovrebbe saper analizzare e progettare la sua soluzione, creando codice semplice e manutenibile. Questo non è un libro sulla progettazione, ma cercheremo di stuzzicare la curiosità del lettore sull'argomento per cercare di farne comprendere l'importanza.

7.1 Ereditarietà

Come tutti i paradigmi che caratterizzano l'Object Orientation, anche il concetto di **ereditarietà** è ispirato a qualcosa che esiste nella realtà. Il termine è inteso in senso darwiniano. Nel mondo reale noi classifichiamo tutto con classi e sottoclassi. Per esempio un cane è un animale, un aereo è un veicolo, la chitarra è uno strumento musicale. In Java l'ereditarietà è la caratteristica che mette in relazione di estensibilità più classi che hanno caratteristiche comuni. Per esempio la classe `Cane` estenderà la classe `Animale`. Il risultato immediato è la possibilità di ereditare codice già scritto, e quindi di gestire insiemi di classi collettivamente giacché accomunate da alcune caratteristiche. Le regole per utilizzare correttamente l'ereditarietà sono semplici e chiare. Ma sebbene l'ereditarietà sia un argomento agevole da comprendere, non è sempre utilizzata in maniera corretta.

7.1.1 La parola chiave `extends`

La parola chiave **`extends`** ci permetterà di implementare l'ereditarietà. Consideriamo le seguenti classi che non incapsuliamo per semplicità:

```
public class Libro {
    public String titolo;
    public String autore;
    public String editore;
    public int numeroPagine;
    public int prezzo;
    //...
}
public class LibroSuJava {
    public String titolo;
    public String autore;
    public String editore;
    public int numeroPagine;
```

```
    public int prezzo;
    public final String ARGOMENTO_TRATTATO = "Java";
    //...
}
```

Notiamo che le classi `Libro` e `LibroSuJava` rappresentano due concetti in relazione tra loro e quindi dichiarano campi in comune. L'ereditarietà consentirà di mettere in relazione di estensione le due classi con la seguente sintassi:

```
public class LibroSuJava extends Libro {
    public final String ARGOMENTO_TRATTATO = "Java";
    //...
}
```

In questo modo la classe `LibroSuJava` erediterà tutti i campi pubblici della classe che estende. Quindi nella classe `LibroSuJava` sono presenti anche le variabili pubbliche `numeroPagine`, `prezzo`, `titolo`, `autore` ed `editore` definite nella classe `Libro`, anche se non sono state codificate esplicitamente. La costante `ARGOMENTO_TRATTATO` invece, appartiene solo alla classe `LibroSuJava`. In particolare diremo che `LibroSuJava` è **sottoclasse** di `Libro`, e `Libro` è **superclasse** di `LibroSuJava`. Il primo vantaggio è evidente, in questo esempio abbiamo riusato del codice evitando un "copia-incolla", ma l'ereditarietà non si riduce al solo scrivere meno righe.

La notazione UML che definisce la relazione di estensione, è una freccia che va dalla sottoclasse alla superclasse. Nella figura 7.1 è rappresentato un diagramma delle classi che mostra la relazione di ereditarietà tra le due classi.

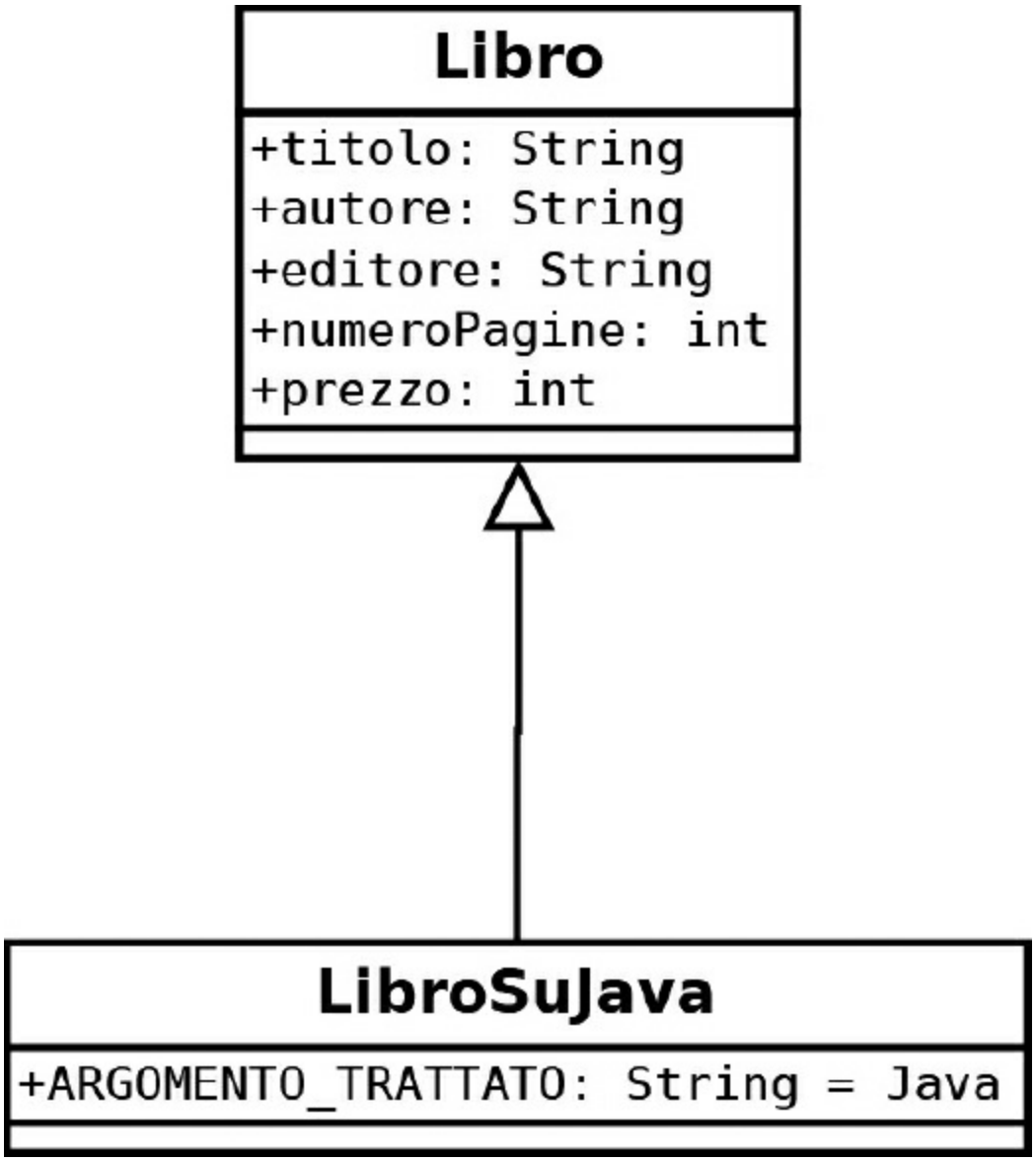


Figura 7.1 - Class Diagram "Ereditarietà: LibroSuJava è sottoclasse di Libro".

7.2 Il modificatore `final`

Abbiamo già visto come questo modificatore venga usato per dichiarare le costanti in Java. Ma **`final`** può essere utilizzato non solo con le variabili, ma anche con classi e metodi per influire sulle regole dell'ereditarietà. Potremmo tradurre il termine `final` proprio con "finale", nel senso di "non modificabile". Infatti:

- una variabile (sia d'istanza che locale) dichiarata `final` diviene una

costante;

- un metodo dichiarato `final` non può essere riscritto in una sottoclasse (non è possibile applicare l'override). Questo significa che si eredita così come è stato dichiarato nella superclasse;
- una classe dichiarata `final` non può essere estesa.

Per esempio le classi `String` e la classe `Math` del package `java.lang` sono esempi di classi dichiarate `final`. Queste classi quindi non si possono estendere perché otterremmo un errore in compilazione come il seguente:

```
jshell> class FinalClassExtendsTest extends String {}
|  Error:
|  cannot inherit from final java.lang.String
|  class FinalClassExtendsTest extends String {}
|                                      ^----^
```

Il modificatore `final` si può utilizzare anche per variabili locali e parametri locali di metodi. In tali casi, i valori di tali variabili non saranno modificabili localmente. Per esempio, il seguente codice viene compilato senza errori:

```
public class LocalVariables {
    public static void main(String args[]) {
        System.out.println(
          new
LocalVariables().finalLocalVariablesMethod(5,6));
    }
    public int finalLocalVariablesMethod(final int i,final
int j) {
        final int k = i + j;
        return k;
    }
}
```

Dichiarare `final` i parametri di un metodo è considerato da alcuni come una buona pratica di programmazione. Il modificatore in questo caso enfatizza il fatto che i parametri non debbano essere modificati. Infatti essi vengono valorizzati dall'esterno, e cambiare loro il valore all'interno del metodo significa snaturare la loro definizione.

7.3 La classe `Object`

Come abbiamo già osservato più volte, il compilatore Java inserisce spesso, nel bytecode compilato, alcune istruzioni che il programmatore non ha inserito. Questo al fine di agevolare lo sviluppatore sia nell'apprendimento che nella codifica. Abbiamo inoltre già asserito più volte che la programmazione ad oggetti si ispira a concetti reali. Tutta la libreria standard di Java è stata pensata ed organizzata in maniera tale da soddisfare la teoria degli oggetti. Siccome la realtà è composta da oggetti (tutto può considerarsi un oggetto, sia elementi concretamente esistenti, sia concetti astratti), nella libreria standard di Java esiste una classe chiamata **`Object`** che astrae il concetto di *oggetto generico*. Esso appartiene al package `java.lang` ed è di fatto la superclasse di ogni classe. È in cima alla gerarchia delle classi e quindi tutte le classi ereditano i membri di `Object` (è possibile verificare quest'affermazione leggendo la descrizione dei metodi di una qualsiasi classe nella documentazione). Se definiamo una classe che non estende altre classi, essa automaticamente estenderà `Object`. Ciò significa che se scrivessimo:

```
public class Arte {
    //...
}
```

il compilatore tradurrebbe questa classe nel seguente modo:

```
public class Arte extends Object {
    //...
}
```

> **Nel mondo reale tutto è un oggetto, quindi in Java tutte le classi estenderanno** `Object`.

Quindi, quando creiamo una nostra classe, questa viene inserita nella gerarchia di classi alla cui cima c'è `Object`. Ogni classe quindi eredita tutti i metodi definiti dalla classe `Object`, e questi verranno spiegati nei prossimi capitoli.

7.4 Rapporto ereditarietà-incapsulamento

Dal momento che l'incapsulamento si può praticamente considerare obbligatorio e l'ereditarietà un prezioso strumento di sviluppo, bisognerà chiedersi che cosa provocherà l'utilizzo combinato di entrambi i paradigmi. Ovvero: che cosa erediteremo da una classe incapsulata? Abbiamo già affermato alla fine del capitolo precedente che estendere una classe significa ereditarne i membri non privati. Consideriamo quindi una classe `Ricorrenza` ottenuta specializzando la classe `Data` definita nel capitolo precedente come esempio di incapsulamento. È escluso che la classe `Ricorrenza` possa accedere alle variabili `giorno`, `mese` e `anno` direttamente, giacché queste non saranno ereditate. Infatti tali variabili erano state dichiarate private all'interno della superclasse `Data`. Ma essendo tutti i metodi set e get dichiarati pubblici nella superclasse, verranno ereditati e quindi resi utilizzabili nella sottoclasse come mostrato in figura 7.2.

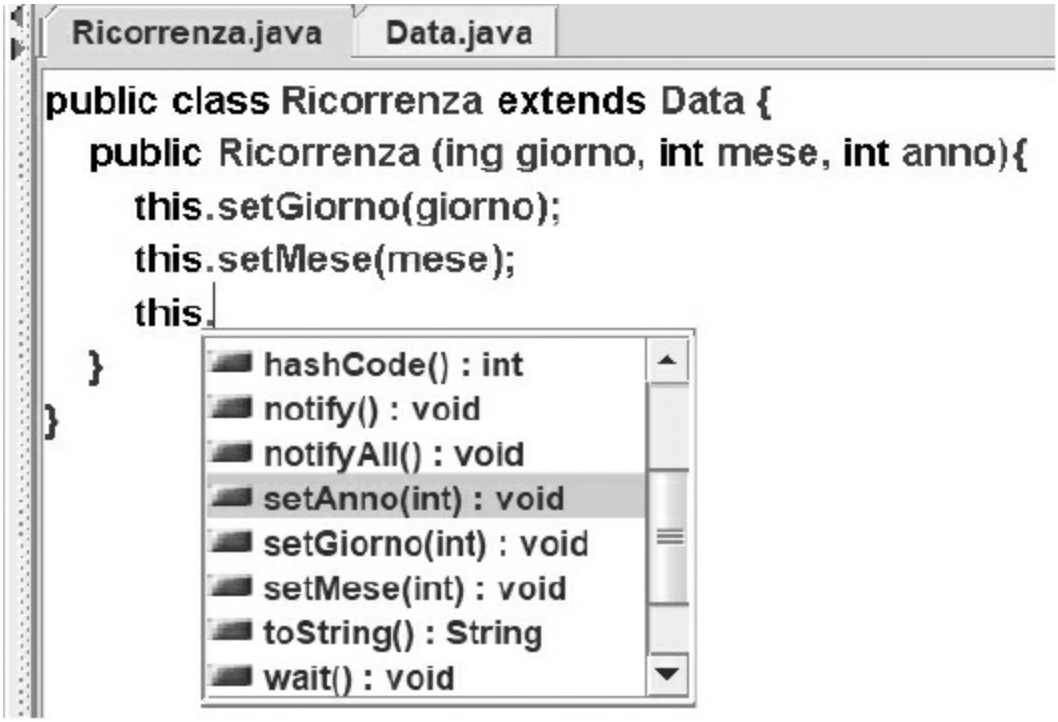


Figura 7.2 - EJE: Classe Ricorrenza che accede ai membri pubblici ereditati.

Quindi anche se la classe `Ricorrenza` non possiede esplicitamente le variabili private di `Data`, è come se le possedesse, visto che può comunque usufruirne tramite l'incapsulamento.

Un errore comune è quello di pensare che per poter utilizzare una certa variabile in una sottoclasse, sia necessario dichiararla `protected`, quando è più che sufficiente avere a disposizione i metodi mutator (set) ed accessor (get) nelle sottoclassi.

Attenzione che, inoltre, dichiarare protetta una variabile d'istanza significa renderla pubblica a tutte le classi dello stesso package (cfr. paragrafo 6.7). Questo significa che la variabile `protected` è accessibile direttamente a tutte le classi appartenenti allo stesso package, e tranne in rari casi non è questo il livello di incapsulamento desiderato.

7.5 Quando utilizzare l'ereditarietà

Quando si parla di ereditarietà si è spesso convinti che per implementarla basti avere un paio di classi che dichiarino campi in comune. Ciò potrebbe essere interpretato solo come un indizio di ereditarietà. Il test decisivo deve però essere effettuato mediante la cosiddetta **"is a" relationship** (in italiano la **relazione "è un").**

7.5.1 La relazione "is a"

Per poter fare un uso corretto dell'ereditarietà, il programmatore deve porsi solo una semplice domanda: "un oggetto della candidata sottoclasse **è un** oggetto della candidata superclasse?".

Se la risposta alla domanda è positiva, l'ereditarietà si deve applicare, se la risposta è negativa, l'ereditarietà non si deve applicare.

Effettivamente, se l'applicazione dell'ereditarietà dipendesse solamente dai campi in comune tra due classi, potremmo trovare relazioni d'estensione tra classi quali `Triangolo` e `Rettangolo`. Per esempio:

```
public class Triangolo {
    public final int NUMERO_LATI = 3;
    public float lunghezzaLatoUno;
    public float lunghezzaLatoDue;
    public float lunghezzaLatoTre;
    //...
}

public class Trapezio extends Triangolo {
```

```
    public final int NUMERO_LATI = 4;
    public float lunghezzaLatoQuattro;
    //...
}
```

Ma un trapezio non è un triangolo! E per la relazione “is a” quest’estensione non è valida. Il problema è che la costruzione di un programma avviene passo dopo passo. Se mettiamo in relazione di ereditarietà in maniera errata due classi, in un primo momento risparmieremmo del codice, ma i problemi sorgerebbero dopo, quando tenteremo di sfruttare un trapezio così come sfruttiamo un triangolo. La relazione “è un” ci mette al sicuro da questi inconvenienti, ed è molto semplice da testare.

Notare che i problemi logici di un’astrazione scorretta dei dati potrebbero essere propagati al codice implementando ereditarietà e incapsulamento. On line trovate l’appendice G intitolata “Un esempio guidato alla programmazione ad oggetti” che dimostra la veridicità dell’ultima affermazione.

L’ereditarietà è quindi definita dalla relazione “è un”, la quale si applica agli oggetti, non alle classi.

7.5.2 Ereditarietà e costruttori

L’ereditarietà non è applicabile ai costruttori. Anche quando sono dichiarati pubblici, i costruttori non sono ereditati per un motivo molto semplice: il loro nome. Ricordiamo che un costruttore è stato definito nel capitolo 2 come metodo speciale, in quanto possiede le seguenti proprietà:

1. ha lo stesso nome della classe cui appartiene;
2. non ha tipo di ritorno;
3. è chiamato automaticamente (e solamente) ogni volta che viene istanziato un oggetto della classe cui appartiene, relativamente a quell’oggetto;
4. è presente in ogni classe.

Relativamente all'ultimo punto abbiamo anche definito il "costruttore di default" come il costruttore che è introdotto in una classe dal compilatore al momento della compilazione, nel caso il programmatore non gliene abbia fornito uno in modo esplicito. Abbiamo anche affermato che solitamente un costruttore è utilizzato per inizializzare le variabili degli oggetti al momento dell'istanza. Per quanto detto, se per esempio la classe `LibroSuJava` ereditasse dalla classe `Libro` un costruttore, erediterebbe un costruttore che si chiama proprio `Libro()`. Un costruttore che si chiama `Libro()` in una classe che si chiama `LibroSuJava` non potrà mai essere chiamato! Infatti per istanziare un oggetto dalla classe `LibroSuJava`, bisogna obbligatoriamente chiamare un costruttore chiamato `LibroSuJava()`. Per esempio, dato il seguente codice:

```
  LibroSuJava libroSuJava = new LibroSuJava();
```

Se al suo posto scrivessimo:

```
  LibroSuJava libroSuJava = new Libro();
```

istanzieremmo un oggetto della classe `Libro`, e non un oggetto della classe `LibroSuJava` (ottenendo anche un errore in compilazione).

Il fatto che i costruttori non siano ereditati dalle sottoclassi è assolutamente in linea con la sintassi del linguaggio, ma contemporaneamente è in contraddizione con i principi della programmazione ad oggetti. In particolare sembra violata la regola dell'astrazione. Infatti, nel momento in cui lo sviluppatore ha deciso di implementare il meccanismo dell'ereditarietà, ha dovuto testarne la validità mediante la cosiddetta relazione "is a". Alla domanda: "un oggetto istanziato dalla candidata sottoclasse può considerarsi anche un oggetto della candidata superclasse?" ha infatti risposto affermativamente.

Un libro su Java, essendo anche un libro, deve avere tutte le caratteristiche di un libro. In particolare deve riutilizzarne anche il costruttore. Non potendolo ereditare però, l'astrazione sembra violata. Invece è proprio in una situazione del genere che Java dimostra la sua coerenza. Aggiungiamo infatti un'altra proprietà alla definizione di costruttore:

5. un qualsiasi costruttore (anche quello di default), come prima

istruzione, invoca sempre un costruttore della superclasse.

Per esempio, aggiungiamo dei costruttori alle classi `Libro` e `LibroSuJava`:

```
public class Libro {
    //...
    public Libro() {
        System.out.println("Costruito un Libro!");
    }
}
public class LibroSuJava extends Libro {
    public LibroSuJava() {
        System.out.println("Costruito un Libro su Java!");
    }
}
```

Il lettore, avendo appreso che i costruttori non sono ereditati, dovrebbe concludere che l'istanza di un `LibroSuJava`, mediante una sintassi del tipo:

```
new LibroSuJava(); /* N.B.: L'assegnazione di un reference
non è
 obbligatoria per istanziare un oggetto */
```

produrrebbe in output la seguente stringa:

```
Costruito un Libro!
```

L'output risultante sarà invece:

```
Costruito un Libro!
Costruito un Libro su Java!
```

Infatti il costruttore `LibroSuJava()` ha prima invocato il costruttore della superclasse `Libro()` e poi è stato eseguito.

La chiamata obbligatoria ad un costruttore di una superclasse viene effettuata tramite la parola chiave `super`, che viene introdotta di seguito.

7.5.3 La parola chiave super

Nel capitolo 6 abbiamo definito la parola chiave `this` come "reference implicito all'oggetto corrente". Possiamo definire la parola chiave **`super`** come "reference implicito all'intersezione tra l'oggetto corrente e la sua

superclasse". Questo reference ci permette di accedere ai componenti della superclasse ed in particolare al suo costruttore.

La parola chiave `super` è strettamente legata al concetto di costruttore. In ogni costruttore, infatti, è sempre presente una chiamata al costruttore della superclasse, tramite una sintassi speciale che sfrutta il reference `super`. Per esempio nella classe `LibroSuJava`, il costruttore verrà modificato dal compilatore nel seguente modo:

```
public class LibroSuJava extends Libro {
    public LibroSuJava() {
        super(); //istruzione implicita se non fornita
esplicitamente.
        System.out.println("Costruito un Libro su Java!");
    }
}
```

Ecco come il costruttore della classe `Libro` viene invocato dal costruttore della classe `LibroSuJava`. Possiamo esplicitare la chiamata a `super()`, ma se non lo facciamo il compilatore considererà questa istruzione implicita. La chiamata ad un costruttore della superclasse è quindi inevitabile!

> **Notare che anche il costruttore della classe `Libro` chiamerà il costruttore della sua superclasse `Object`, mediante un comando `super()` implicito.**

Supponiamo ora di voler dotare la superclasse `Libro` di un costruttore che imposti la variabile `titolo`. D'altronde un oggetto di tipo `Libro` almeno un titolo lo dovrebbe avere:

```
public class Libro {
    //...
    public Libro(String titolo) {
        this.titolo = titolo;
    }
}
```

Questa classe compilerà, ma `LibroSuJava` non compilerà più:

```
LibroSuJava.java:1: error: constructor Libro in class Libro
  cannot be applied to given types;
public class LibroSuJava extends Libro {
       ^
  required: String
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Modifichiamo la superclasse e non compila la sottoclasse, strano! Ma neanche poi tanto visto che sono legate dalla relazione “is a”. Leggendo l’errore del compilatore comunque si può intuire quello che è successo. Il costruttore della sottoclasse ha cercato di chiamare un costruttore della superclasse che non c’è: il costruttore senza parametri. Infatti, se un costruttore viene aggiunto esplicitamente, nessun altro costruttore di default viene inserito implicitamente. Quindi l’istruzione `super()` inserita automaticamente nella sottoclasse `LibroSuJava`, prova a chiamare il costruttore della superclasse senza parametri (infatti non sono passati parametri tra le parentesi). Per risolvere il problema, bisogna modificare il costruttore di `LibroSuJava` in modo tale da fargli chiamare il costruttore della superclasse `Libro` che prende in input il titolo. La soluzione seguente è la migliore:

```
public class LibroSuJava extends Libro {
    public LibroSuJava(String titolo) {
        super(titolo);
    }
    //...
}
```

Abbiamo quindi inserito una chiamata esplicita al costruttore della superclasse che prende in input una stringa. Ci è venuto naturale modificare anche la lista dei parametri della sottoclasse, ed anche questo può essere considerato indice della coerenza di Java verso i paradigmi object oriented.

Attenzione: la chiamata al costruttore della superclasse mediante `super()` deve essere la prima istruzione di un costruttore e non potrà essere inserita all’interno di un metodo che non sia un costruttore.

Se avessimo più costruttori nella superclasse potremmo scegliere quale chiamare. Per esempio se la classe `Libro` definisse più costruttori nel seguente modo:

```
public class Libro {
    //...
    public Libro(String titolo, String autore) {
        this(titolo); // Chiamata al secondo costruttore
        setAutore(autore);
    }
    public Libro(String titolo) {
        this.titolo = titolo;
    }
    //...
}
```

La classe `LibroSuJava` potrebbe chiamare a seconda del caso il costruttore più adeguato. Per esempio:

```
public class LibroSuJava extends Libro {
    public LibroSuJava(String titolo) {
        super(titolo);
    }

    public LibroSuJava(String titolo, String autore) {
        super(titolo, autore);
    }
    //...
}
```

La chiamata tramite `super()` ad un costruttore di una superclasse o tramite `this()`ad un costruttore della classe stessa, può essere usata solo come prima istruzione in un costruttore. Se una di queste istruzioni non fosse inserita come prima istruzione, il compilatore segnalerebbe un errore. Questo implica che solo un'istruzione tra `super()` o `this()` sarà presente in un costruttore. Se si inserisce esplicitamente il comando `this()`, allora la chiamata a `super()` non può essere inserita nello stesso costruttore. Notare però che in casi come questo, un costruttore chiama un altro costruttore mediante il comando `this()`. Quest'altro costruttore a sua volta o chiama un terzo

costruttore tramite un altro comando `this()`, o chiama un costruttore della superclasse con il comando `super()`! Insomma il costruttore della superclasse verrà chiamato in qualsiasi caso prima o poi. Per esempio la seguente classe che rappresenta un carattere (font) da utilizzare in un editor:

```
public class Carattere {
    private String tipo;
    private int dimensione;

    public Carattere(String tipo) {
        this(tipo, 12);
    }

    public Carattere(String tipo, int dimensione) {
        //qui verrà inserito un super() dal compilatore
        setTipo(tipo);
        setDimensione(dimensione);
    }
    //Omessi metodi set e get...
}
```

dichiara un primo costruttore che invoca il secondo con il comando `this()`. Nel secondo costruttore però, il compilatore inserirà esplicitamente il comando `super()` come prima istruzione. La chiamata ad un costruttore della superclasse (che in questo caso è la classe `Object`) è stata solo rimandata.

> **Non è neanche possibile chiamare un costruttore tramite i comandi `super()` o `this()` da un metodo che non sia un costruttore.**

Visto che `super` e `this` possono addirittura chiamare costruttori, possibile che non possano chiamare metodi? Certo che possono farlo! Per il `this` lo abbiamo già visto nel capitolo precedente. Con la sintassi:

```
this.nomeMetodo()
```

era possibile chiamare metodi della stessa classe; ma avevamo detto che il reference `this` in questi casi è ridondante, visto che Java lo considera implicito se non è specificato un reference prima della chiamata di un

metodo. Ovvero scrivere:

```
this.nomeMetodo()
```

è equivalente a scrivere:

```
nomeMetodo()
```

Invece con la sintassi:

```
super.nomeMetodo()
```

sarà possibile invocare un metodo di una superclasse da un metodo di una sottoclasse. Questo ha senso in caso di riscrittura di un metodo in una sottoclasse, altrimenti il metodo della superclasse sarebbe comunque ereditato anche nella sottoclasse, e quindi sarebbe inutile utilizzare `super`. Per esempio consideriamo le seguenti classi:

```
public class Persona {
    private String nome, cognome;
    public String toString() {
        return nome + " " + cognome;
    }
    //accessor e mutator methods (set e get)
}

public class Cliente extends Persona {
    private String indirizzo, telefono;
    public String toString() {
        return super.toString()+ "\n" + indirizzo + "\nTel:"
+ telefono;
    }
    //accessor e mutator methods (set e get)
}
```

La classe `Persona` dichiara un metodo che si chiama `toString()`. Si tratta di uno dei metodi più famosi, in quanto dichiarato direttamente dalla classe `Object`, e dovrebbe essere il metodo responsabile a rappresentare con una stringa l'oggetto su cui viene chiamato il metodo. Quindi la classe `Persona` sta già facendo **override (riscrittura)** del metodo, facendo sì che il metodo ritorni una stringa composta dal `nome` e dal `cognome`.

La classe `Cliente` estende la classe `Persona` e ne eredita anche il metodo

`toString()`. Ma questo metodo viene a sua volta ridefinito per stampare anche i nuovi campi definiti nella classe `Cliente: indirizzo` e `telefono`. Per stampare anche il `nome` e il `cognome`, il metodo `toString()` della classe `Cliente` in primo luogo chiama il metodo `toString()` della superclasse `Persona` (riuso per evitare duplicazioni di codice) e poi concatena il risultato con i dettagli di indirizzo e telefono. Ricordiamo che la stringa `\n` equivale ad andare a capo come abbiamo visto nel capitolo 3.

> **Se il metodo `toString()` non fosse stato riscritto, avrebbe restituito una stringa con il formato `NomeClasse@IndirizzoInFormatoEsadecimale`. Questo perché essendo la classe `Object` molto generica, i creatori di Java decisero di restituire una stringa descrittiva del reference. L'override sarà ampiamente approfondito nel prossimo capitolo.**

7.5.4 Ereditarietà e modificatori

Riassumiamo la situazione per quanto riguarda i modificatori d'accesso.

Il modificatore `private` può essere applicato a variabili e metodi, e impedisce a tali membri di essere ereditati in una sottoclasse.

Il modificatore `protected`, che pure può essere applicato solo a variabili e metodi, permette che i membri siano ereditati. Per esempio consideriamo la seguente classe (già incontrata nel capitolo precedente) che dichiara una variabile e un metodo protetti:

```
package com.cdsc.test;

public class ProtectedInClasse {
    protected int variabileProtected;

    protected void metodoProtected() {
        System.out.println("Invocato metodo protected");
    }
}
```

La seguente sottoclasse, pur appartenendo ad un package diverso, eredita ed

usa i membri `protected`:

```
package com.cdsc;

import com.cdsc.test.*;

public class ProtectedSottoclasseFuoriPackage extends
ProtectedInClasse {
    public void metodoCheUsaProtected() {
        metodoProtected();
        System.out.println(variabileProtected);
    }
}
```

Anche una sottoclasse appartenente allo stesso package eredita e può usare i membri `protected`:

```
package com.cdsc.test;

public class ProtectedSottoclasseInPackage extends
ProtectedInClasse {
    public void metodoCheUsaProtected() {
        metodoProtected();
        System.out.println(variabileProtected);
    }
}
```

Quando invece non specifichiamo modificatori davanti a variabili d'istanza o metodi, questi saranno ereditati solo nelle sottoclassi appartenenti allo stesso package. Invece, se non specifichiamo modificatori quando dichiariamo una classe, essa sarà visibile (e quindi estendibile) solo da classi appartenenti allo stesso package. Infatti, se dichiariamo la classe:

```
package com.cdsc.test;

/* public */ class SuperclasseInPackage {
}
```

la sottoclasse seguente non compilerà se non dichiariamo `public` la superclasse `TestPackageEreditarieta`:

```
package com.cdsc;
```

```
import com.cdsc.test.*;

public class SottoclasseInPackageDiverso extends
TestPackageEreditarieta {

}
```

Di seguito l'output:

```
SottoclasseInPackageDiverso.java:5: error: cannot find symbol
public class SottoclasseInPackageDiverso extends
SuperclasseInPackage {
                                                ^
  symbol: class SuperclasseInPackage
1 error
```

Invece la seguente classe compilerà perché appartenente allo stesso package:

```
package com.cdsc.test;

public class SottoclasseInPackage extends
SuperclasseInPackage {
}
```

Abbiamo già parlato del modificatore `final` e del suo rapporto con l'ereditarietà. Riassumendo una classe dichiarata `final` non può essere estesa, un metodo dichiarato `final` non può essere riscritto (override) in una sottoclasse, una variabile `final` diventa una costante. È però possibile ereditare una costante (a meno che non sia dichiarata `private`).

Il modificatore `static` può essere utilizzato con il comando `import`, per importare membri statici di una classe, ma soprattutto può essere applicato a variabili, metodi ed anche inizializzatori. Il comando `import` non ha nulla a che fare con l'ereditarietà. Invece variabili e metodi statici, vengono ereditati, ma siccome appartengono alla classe e non agli oggetti, si tratta di un**'ereditarietà tra classi.**

Ci sono delle differenze e delle eccezioni con la classica ereditarietà. Per esempio consideriamo la classe:

```
public class ClasseConMetodoStatico {
    public static void metodoStatico() {
        System.out.println("Questo è un metodo statico");
```

```
    }
}
```

È possibile estenderla ed utilizzare `metodoStatico()` senza referenziarlo in questo modo:

```
public class SottoclasseConMetodoStatico extends
ClasseConMetodoStatico {
    public void metodoQualsiasi() {
        metodoStatico();
    }
}
```

Possiamo anche verificare l'ereditarietà per classi con la classe:

```
public class TestClasseConMetodoStatico {
    public static void main(String args[]) {
        SottoclasseConMetodoStatico.metodoStatico();
    }
}
```

Tratteremo a parte il modificatore `abstract` nel paragrafo 7.6.

Gli altri modificatori come `volatile`, `synchronized`, `transient`, `strictfp`, `native` non hanno particolari proprietà relativamente all'ereditarietà.

7.5.5 Ereditarietà e inizializzatori

Anche gli inizializzatori statici e d'istanza sembra che vengano in qualche modo ereditati, ma non si tratta di ereditarietà vera e propria, semmai del normale flusso di esecuzione. Dimostriamolo direttamente con un esempio. Consideriamo una classe che definisce due semplici inizializzatori (uno statico e uno d'istanza), oltre ad un costruttore senza parametri:

```
public class ParentInitializer {
    public ParentInitializer() {
        System.out.println("Costruttore di
ParentInitializer");
    }
    {
        System.out.println("Inizializzatore d'istanza di
ParentInizializer");
```

```
    }

static {
    System.out.println("Inizializzatore statico di
ParentInizializer");
    }
}
```

Estendiamo `ParentInitializer` con la classe `ChildInitializer` che dichiara a sua volta altri due inizializzatori e un metodo `main()` che la auto-istanzia:

```
public class ChildInitializer extends ParentInitializer {
    public ChildInitializer() {
        System.out.println("Costruttore di
ChildInitializer");
    }
    {
        System.out.println("Inizializzatore d'istanza di
ChildInizializer");
    }

    static {
        System.out.println("Inizializzatore statico di
ChildInizializer");
    }
    public static void main(String args[]) {
        new ChildInitializer();
    }
}
```

L’output risultante sarà il seguente:

```
Inizializzatore statico di ParentInizializer
Inizializzatore statico di ChildInizializer
Inizializzatore d'istanza di ParentInizializer
Costruttore di ParentInitializer
Inizializzatore d'istanza di ChildInizializer
Costruttore di ChildInitializer
```

Si noti che, per gli inizializzatori sia statici sia d’istanza, non valgono i modificatori d’accesso, e quindi non valgono le regole dell’ereditarietà. Quello che accade è che prima che sia caricata in memoria la classe `ChildInitializer`, deve essere caricata in memoria la classe

`ParentInitializer`, e questo spiega le prime due righe dell'output. Ricordiamo che mentre gli inizializzatori statici sono invocati al momento del caricamento in memoria della classe, gli inizializzatori d'istanza sono chiamati al momento dell'istanza dell'oggetto, prima del costruttore. Nell'esempio capita che sia chiamato il costruttore di `ChildInitializer` che come abbiamo visto chiama il costruttore di `ParentInitializer`. Ma prima che questo sia eseguito, viene eseguito l'inizializzatore di `ParentInitializer`. Lo stesso accade nella sottoclasse, dove prima che sia chiamato il costruttore viene invocato l'inizializzatore, ed ecco spiegato l'output del precedente esempio.

Gli inizializzatori si usano raramente, ma è bene sapere come sono eseguiti in caso di ereditarietà.

7.5.6 Generalizzazione e specializzazione

Sono due termini che definiscono i processi che portano all'implementazione dell'ereditarietà.

Si parla di **generalizzazione** se, a partire da un certo numero di classi, si definisce una superclasse che ne raccoglie le caratteristiche comuni. Viceversa si parla di **specializzazione** quando partendo da una classe, si definiscono una o più sottoclassi allo scopo di ottenere oggetti più specializzati.

L'utilità della specializzazione è semplice da evidenziare: supponiamo di voler creare una classe `MioBottone`, le cui istanze siano visualizzate come bottoni da utilizzare su un'interfaccia grafica. Partire da zero creando pixel per pixel è molto complicato. Se invece estendiamo la classe `Button` del package `java.awt`, dobbiamo solo aggiungere il codice che personalizzerà il `MioBottone`.

Nell'ultimo esempio del paragrafo 7.5.1 avevamo a disposizione la classe `Triangolo` e la classe `Trapezio`. Abbiamo notato come il test "is a", fallendo, ci sconsigli l'implementazione dell'ereditarietà. Eppure queste due classi hanno campi in comune e non sembra che sia un evento casuale. In effetti sia il `Triangolo` sia il `Trapezio` sono ("is a") entrambi poligoni. La

soluzione a questo problema è *naturale*. Basta generalizzare le due astrazioni in una classe `Poligono`, che potrebbe essere estesa dalle classi `Triangolo` e `Rettangolo`. Per esempio:

```
public class Poligono {
    public int numeroLati;
    public float lunghezzaLatoUno;
    public float lunghezzaLatoDue;
    public float lunghezzaLatoTre;
    //...
}

public class Triangolo extends Poligono {
    public final int NUMERO_LATI = 3;
    public Triangolo() {}
    //...
}

public class Trapezio extends Poligono {
    public final int NUMERO_LATI = 4;
    public float lunghezzaLatoQuattro;
    //...
}
```

Se fossimo partiti dalla classe `Poligono` per poi definire le due sottoclassi, avremmo parlato invece di specializzazione. Usando il processo di specializzazione è facile pensare anche ad altre eventuali sottoclassi come `Esagono`, `Pentagono`, `Ottagono`, `Quadrato` e così via. La classe `Poligono` quindi è la più generica, e in un eventuale programma che usa poligoni, l'unica classe che probabilmente non istanzieremo sarà proprio la classe `Poligono`. In questi casi la classe `Poligono` va dichiarata `abstract`, e questo implicherà che non potrà essere istanziata (otteremmo un errore in compilazione se ci provassimo). Quindi `Poligono` rimarrebbe importante solo per permetterci di sfruttare i vantaggi dell'ereditarietà, ma non esisteranno oggetti istanziati direttamente da questa classe. Tuttavia, per quanto detto, oggetti istanziati dalle sue sottoclassi `Trapezio` e `Triangolo` potranno essere considerati tutti poligoni!

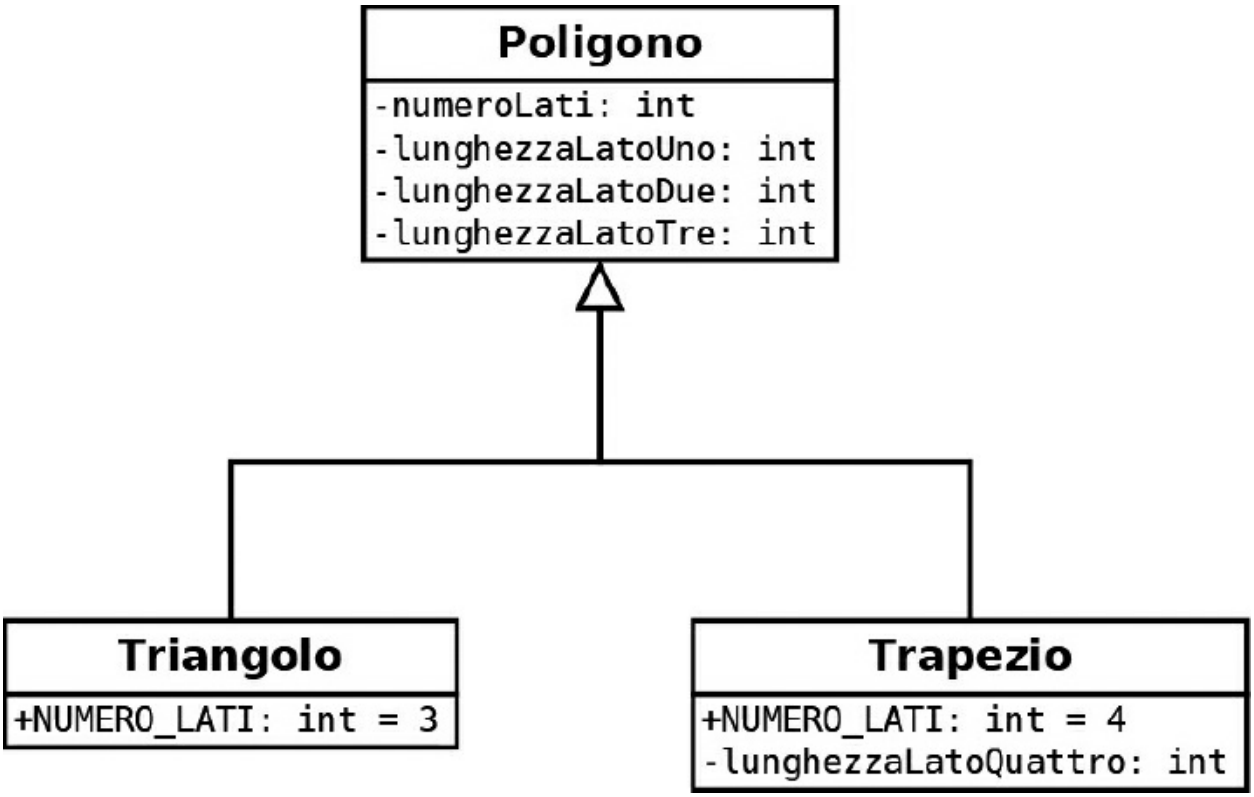


Figura 7.3 - Class Diagram della gerarchia dei poligoni.

Nella figura 7.3 abbiamo rappresentato la gerarchia delle classi presentate. Per semplicità abbiamo omesso di dettagliare i metodi (ricordiamo che UML in questo libro è usato spesso in modo informale).

7.6 Il modificatore `abstract`

Il modificatore **`abstract`** può essere applicato non solo a classi ma anche a metodi. Non ha senso invece parlare di variabili astratte.

7.6.1 Metodi astratti

Un metodo astratto non implementa un proprio blocco di codice e quindi è privo di comportamento. La definizione di un metodo astratto non possiede parentesi graffe ma termina con un punto e virgola. Un esempio di metodo astratto è il seguente:

```
  public abstract void dipingiQuadro();
```

Questo metodo non potrà essere invocato (in quanto non è definito) ma potrà essere soggetto a riscrittura (override) in una sottoclasse come vedremo approfonditamente nel prossimo capitolo dedicato al polimorfismo.

Un metodo astratto potrà essere definito solamente all'interno di una classe astratta. In altre parole, una classe che contiene anche un solo metodo astratto deve essere dichiarata astratta.

7.6.2 Classi astratte

Una classe dichiarata astratta non può essere istanziata. Il programmatore che ha intenzione di marcare una classe con il modificatore `abstract` deve essere consapevole a priori che da quella classe non saranno istanziabili oggetti. Ma perché creare una classe che non si può istanziare? Perché a volte c'è bisogno di creare classi generiche da estendere, e che rappresentano un concetto troppo astratto per essere istanziato nel contesto di un determinato programma. La classe `Poligono` dell'esempio del paragrafo 7.5.6 per esempio, è un'ottima candidata a diventare una classe astratta. Infatti, in un ipotetico utilizzo delle classi che abbiamo definito, non istanzieremo mai oggetti dalla classe `Poligono`, ma solo dalle classi concrete `Triangolo` e `Trapezio`:

```
public class TestPoligoni {
    public static void main(String args[]) {

        Triangolo triangolo = new Triangolo(25, 2, 18);
        Trapezio trapezio = new Trapezio(24, 11, 20, 17);
        System.out.println(triangolo);
        System.out.println(trapezio);
    }
}
```

Quindi possiamo dichiarare la classe `Poligono` astratta come in figura 7.4.

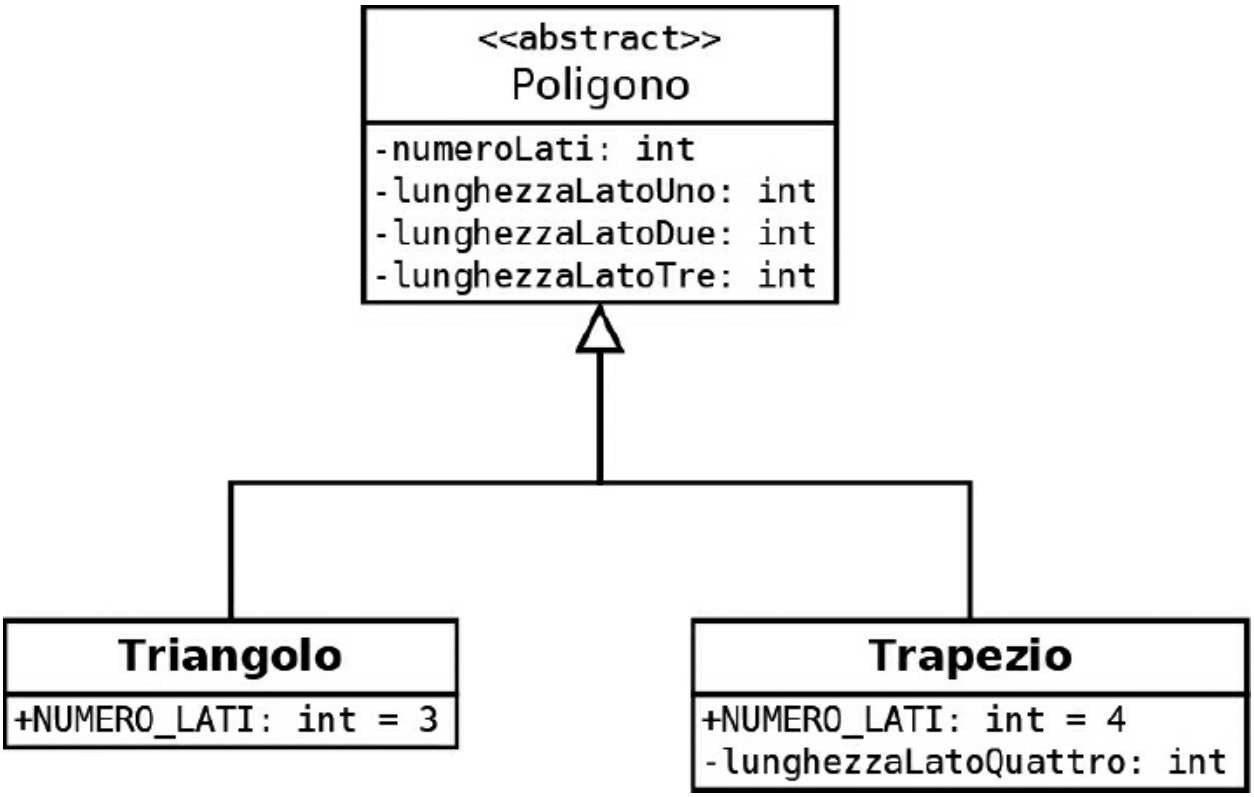


Figura 7.4 - Class Diagram della possibile futura gerarchia dei poligoni.

Si noti che la notazione UML per il modificatore `abstract` è piuttosto criticabile. Infatti basterebbe scrivere il nome dell'entità (classe o metodo) in corsivo. Questa scelta però porta la chiarezza del diagramma a dipendere dal tipo di font scelto che utilizziamo per scrivere il testo (il problema non si pone se per i nostri diagrammi usiamo carta a penna). Infatti non per tutti i font è chiaramente visibile la differenza tra corsivo e non corsivo. Per tale ragione abbiamo aggiunto un cosiddetto stereotipo `<<abstract>>`. Uno stereotipo consente di etichettare un'entità con una parola significativa per caratterizzarlo. Con lo stereotipo `<<abstract>>` abbiamo reso più evidente che la classe `Poligono` è stata dichiarata astratta.

Il modificatore `abstract`, è anche un importante strumento di progettazione. Infatti dichiarando una classe `abstract` dichiareremo anche che questa deve essere estesa, e quindi indirizzeremo lo sviluppo verso la creazione di sottoclassi della classe astratta. Il vantaggio di pensare alla programmazione pensando per superclassi (e poi per interfacce come vedremo nel prossimo paragrafo) piuttosto che perdersi nei dettagli delle sottoclassi, è grande. Infatti l'interazione con il software sarà più semplice se

ragioniamo pensando alle astrazioni piuttosto che alle implementazioni.

Consideriamo la seguente classe astratta che definisce il metodo presentato nel paragrafo precedente:

```
public abstract class Pittore {
    //...
    public abstract void dipingiQuadro();
    //...
}
```

Questa classe ha senso se inserita in un sistema in cui l'oggetto `Pittore` può essere considerato troppo generico per definire un nuovo tipo di dato da istanziare. Supponiamo che per il nostro sistema sia fondamentale conoscere lo stile pittorico di un oggetto `Pittore` e, siccome non esistono pittori capaci di dipingere con un qualsiasi tipo di stile tra tutti quelli esistenti, non ha senso istanziare una classe `Pittore`. Sarebbe corretto popolare il nostro sistema di sottoclassi non astratte di `Pittore` con `PittoreImpressionista` e `PittoreNeoRealista`. Queste sottoclassi devono ridefinire il metodo astratto `dipingiQuadro` (a meno che non si abbia l'intenzione di dichiarare astratte anche esse). Notare che la classe `Pittore` avrebbe potuto implementare concretamente il metodo `dipingiQuadro`, ma a livello logico non sarebbe giusto definirlo favorendo uno stile piuttosto che un altro. Nulla vieta però ad una classe astratta di implementare tutti suoi metodi.

Facciamo un esempio parlando di strumenti musicali:

```
public abstract class Strumento { //Classe astratta
   public String nome;
   public String prezzo;
   public abstract void suonaFaDiesis(); /* Ogni strumento
suona in modo
                                          * diverso!
Impossibile definire
                                          * questo metodo!
                                          */
   //...
}
public class Chitarra extends Strumento { // Classe concreta
    public void suonaFaDiesis() { // Override (riscrittura)
del metodo
        //Implementazione del metodo per la chitarra.
    }
```

```
    //...
}
public abstract class StrumentoAFiato extends Strumento {
// Classe di nuovo astratta che estende Strumento

    //metodo suonaFaDiesis ereditato ancora astratto e non
riscritto
    //perché troppo generico!

    //...
}

public class Flauto extends StrumentoAFiato {
// Classe concreta che estende StrumentoAFiato

   public void suonaFaDiesis() {
       //Implementazione del metodo per il flauto.
   }
   //...
}
```

In questo esempio abbiamo visto come `Strumento` sia la superclasse della classe concreta `Chitarra` (che ridefinisce il metodo `suonaFaDiesis()`) e della classe astratta `StrumentoAFiato` (la quale non ridefinisce il metodo `suonaFaDiesis()` che viene ereditato astratto). Infine la classe concreta `Flauto` estende `StrumentoAFiato` (che a sua volta estendeva `Strumento`). Quindi la chitarra è uno strumento e il flauto è uno strumento a fiato, ma è anche uno strumento: è tutto coerente!

Il grande vantaggio che offre l'implementazione di una classe astratta è che **obbliga** le sue sottoclassi ad implementare un comportamento. A livello di progettazione le classi astratte costituiscono uno strumento fondamentale.

> **Il modificatore** `abstract`, **per quanto riguarda le classi, potrebbe essere considerato l'opposto del modificatore** `final`. **Infatti, una classe** `final` **non può essere estesa, mentre una classe dichiarata** `abstract` **deve essere estesa. Non è ovviamente possibile utilizzare congiuntamente per classi e metodi i modificatori** `abstract` **e** `final`, **per chiari motivi logici. Non è altrettanto possibile utilizzare i**

> **modificatori** `abstract` **e** `static` **congiuntamente per i metodi.**

A differenza del mondo reale, dove una persona potrebbe essere contemporaneamente un genitore, un programmatore, un lettore di libri e un musicista, in Java non esiste propriamente quella che viene definita **ereditarietà multipla:** una classe può estendere solo una classe per volta. Questo significa che una classe `Persona` non può essere dichiarata come nel seguente esempio:

```
public class Persona extends Programmatore, Genitore,
  Lettore, Musicista { // Illegale!
```

Bisognerà scegliere solo una superclasse in quanto in Java dopo la clausola `extends` è possibile specificare solo una classe. Le ragioni per cui non è stata permessa l'ereditarietà multipla in Java risiedono nelle precedenti esperienze di altri linguaggi, i quali hanno dovuto sopperire con stratagemmi complessi ai limiti della definizione. Dalla versione 8 di Java però, il concetto di interfaccia è stato rivisto per supportare anche altre caratteristiche. Oggi con le interfacce è possibile avere una *ereditarietà multipla soft*, che non introduce i problemi presenti in altri linguaggi. Rimane però valida la regola che è possibile estendere una sola classe per volta.

7.7 Interfacce

Dal punto di vista della progettazione, un**'interfaccia** è un'evoluzione del concetto di classe astratta. Dal punto di vista del codice, assomiglia invece ad una classe senza la sua implementazione interna. Un'interfaccia vuole rappresentare infatti quella che con l'incapsulamento abbiamo chiamato proprio *interfaccia pubblica*, ovvero quella parte dell'oggetto visibile all'esterno, che nasconde la sua implementazione interna. Divideremo la spiegazione delle interfacce in due parti, quella pre-Java 8, e quella attuale.

7.7.1 Definizione classica (pre-Java 8)

Sino a Java 7 un'interfaccia per definizione poteva possedere solo metodi

dichiarati implicitamente `public` e `abstract` e variabili dichiarate implicitamente `public`, `static` e `final`. Di seguito un primo semplice esempio di interfaccia:

```
public interface Saluto {
    public static final String CIAO = "Ciao";
    public static final String BUONGIORNO = "Buongiorno";
    ...
    public abstract void saluta();
}
```

Possiamo equivalentemente scrivere quest'interfaccia omettendo tutti i modificatori:

```
public interface Saluto {
    String CIAO = "Ciao";
    String BUONGIORNO = "Buongiorno";
    ...
    void saluta();
}
```

Anche in questo caso Java lavora per noi dietro le quinte aggiungendo i modificatori che non abbiamo scritto. Quindi `CIAO` e `BUONGIORNO` saranno sempre costanti statiche pubbliche, e il metodo `saluta()` sarà comunque implicitamente dichiarato `public` e `abstract` (infatti il metodo termina con un punto e virgola). Questo significa che non è possibile usare modificatori in contrasto con quelli che Java si attende. Per esempio se provassimo a dichiarare il metodo `saluta()` come `protected`, otterremmo un errore in compilazione:

```
Saluto.java:5: error: modifier protected not allowed here
    protected void saluta();
                   ^
1 error
```

In qualsiasi caso ogni membro (variabile o metodo) di una classe astratta ha visibilità pubblica.

Come le classi, le interfacce si devono scrivere all'interno di file che hanno esattamente lo stesso nome dell'interfaccia

che definiscono, e il suffisso .java. Quindi l'interfaccia `Saluto` dell'esempio deve essere salvata all'interno di un file di nome Saluto.java. Anche le convenzioni per i nomi sono le stesse che valgono per le classi.

Un'interfaccia non si può istanziare (non è una classe). Per essere utilizzata, ha bisogno di essere in qualche modo estesa, ma solo un'altra interfaccia può estendere un'interfaccia, e quindi è possibile creare gerarchie costituite da sole interfacce. Dato però che senza istanziare oggetti i programmi non funzionano, abbiamo bisogno che le classi ereditino dalle interfacce. Ecco che allora viene in aiuto la parola chiave `implements`, che si usa in maniera molto simile ad `extends`. La differenza tra implementare un'interfaccia ed estendere una classe, consiste essenzialmente nel fatto che, mentre possiamo estendere una sola classe alla volta, possiamo invece implementare un numero indefinito di interfacce, simulando di fatto l'ereditarietà multipla (che verrà affrontata in dettaglio nei prossimi paragrafi).

Un'interfaccia, essendo un'evoluzione di una classe astratta, non può essere istanziata. Potremmo utilizzare l'interfaccia dell'esempio nel seguente modo:

```
public class SalutoImpl implements Saluto {
    public void saluta() {
        System.out.println(CIAO);
    }
}
```

Riassumendo, una classe può implementare un'interfaccia e un'interfaccia può estendere un'altra interfaccia.

Prima dell'avvento di Java 8, il concetto di interfaccia è stato sempre molto apprezzato per la sua semplicità. Se una classe ereditava metodi da un'interfaccia, allora doveva *onorare il contratto* e ridefinire i metodi astratti ereditati (a meno che non si volesse dichiarare astratta anche la classe). L'uso di un'interfaccia è sempre stato limitato alla sua funzione di essere una specifica astratta per le classi che la implementavano. Ma col passare del tempo i linguaggi di programmazione si sono evoluti per soddisfare i bisogni degli sviluppatori. Negli anni ogni evoluzione di Java ha portato sia benefici

che costi. Dopo Java 8 le interfacce hanno acquisito maggiore potenza, ma contemporaneamente le regole che ne governano l'utilizzo si sono complicate.

7.7.2 Metodi statici

Dalla versione 8 di Java in poi, è possibile definire all'interno delle interfacce anche **metodi statici.** In effetti non c'è mai stata una ragione particolare (come un problema tecnico) per il quale non si era mai voluto inserire metodi statici all'interno delle interfacce. Semplicemente non sembrava in linea con la concezione di interfaccia come contratto da implementare. Oggi quindi è possibile scrivere interfacce nel seguente modo:

```
public interface StaticMethodInterface {
    static void metodoStatico() {
        System.out.println("Metodo Statico Chiamato!");
    }
}
```

E chiamare metodi statici direttamente dalle interfacce:

```
public class TestStaticMethodInterface {
    public static void main(String args[]) {
        StaticMethodInterface.metodoStatico();
    }
}
```

Usando le interfacce contenenti metodi statici ne tradiamo la natura *contrattuale* per la quale fu definita. Un'interfaccia come la precedente non ha bisogno di essere implementata, infatti i metodi statici di un'interfaccia non vengono ereditati. Il comando `implements` quindi non fa ereditare i metodi statici. Interfacce di questo tipo servono semplicemente per definire metodi statici e pubblici, in pratica definiscono funzioni. Per esempio se considerassimo la classe:

```
public class StaticMethodClass implements
StaticMethodInterface {
}
```

allora il codice seguente:

```
public class TestStaticMethodInterface {
    public static void main(String args[]) {
        StaticMethodClass.metodoStatico();
    }
}
```

produrrebbe un errore in compilazione:

```
TestStaticMethodInterface.java:3: error: cannot find symbol
        StaticMethodClass.metodoStatico();
                         ^
  symbol: method metodoStatico()
  location: class StaticMethodClass
1 error
```

In definitiva, bisogna chiamare il metodo sfruttando il nome dell'interfaccia e non il nome della sua sottoclasse:

```
public class TestStaticMethodInterface {
    public static void main(String args[]) {
        StaticMethodInterface.metodoStatico();
    }
}
```

Prima di Java 8 per ottenere strutture con metodi statici, sono sempre state usate le classi, magari astratte o con costruttori privati (come la classe `Math` di cui abbiamo già accennato) o implementate come Singleton (cfr. paragrafo 7.8.6). Usare un'interfaccia popolata di metodi statici in luogo di una classe non sembra essere un grosso vantaggio. Una classe potrebbe dichiarare per esempio un inizializzatore statico per eseguire del codice nel momento in cui è caricata in memoria per la prima volta. In un'interfaccia invece è illegale dichiarare inizializzatori.

Dalla versione 9 di Java è possibile anche creare **metodi statici privati.** I metodi statici privati possono essere chiamati solo e solamente dai metodi statici della stessa interfaccia, e non vengono ereditati. Dobbiamo intenderli semplicemente come

metodi ausiliari per metodi statici.

7.7.3 Metodi di default

Altra novità di Java 8 è la possibilità di dichiarare metodi concreti all'interno delle interfacce. Si parla di **metodi di default** (in inglese **default methods)** perché vengono dichiarati usando come modificatore la parola chiave `default`. Per esempio consideriamo il seguente codice:

```
public interface Solista {
    default void eseguiAssolo() {
        //Scala maggiore in DO
        System.out.println("DO RE MI FA SOL LA SI");
    }
}
```

In questo modo possiamo ereditare questo metodo in un'eventuale sottoclasse senza dover riscrivere il metodo (che non è astratto). La seguente classe compila senza problemi:

```
public class Musicista implements Solista {

}
```

Il metodo `eseguiAssolo()` si può sempre riscrivere all'occorrenza (ma questo è un argomento che approfondiremo nel prossimo capitolo). Avere però un'implementazione di default può essere in generale molto comodo.

È possibile scrivere più di un metodo di default in un'interfaccia, e questi possono quindi convivere con metodi astratti e metodi statici.

Dalla versione 9 di Java è possibile creare anche **metodi privati** ausiliari nelle interfacce. Come nel caso dei metodi privati statici, i metodi privati possono essere invocati solo dai metodi di default dichiarati nella stessa interfaccia. Non bisogna dichiararli come metodi di default, appunto perché sono solo da intendere come metodi ausiliari e non vengono ereditati. Infatti il seguente codice:

```
public interface TestMetodiPrivatiInInterfaccia {
    default void metodoDiDefault() {
        metodoPrivato();
    }

    private default void metodoPrivato() {
        System.out.println("Invocato metodo privato");
    }
}
```

fallirà in compilazione con il seguente errore:

```
TestMetodiPrivatiInI
nterfaccia.java:6: error: illegal combination
  of modifiers: private and default
    private default void metodoPrivato() {
                         ^
1 error
```

7.7.4 Interfacce funzionali

Il nome “interfaccia” in un certo senso ha perso il significato originario, anche se è sempre possibile usare le interfacce dichiarando solo metodi astratti. Con l’introduzione dei metodi privati infatti, le interfacce ormai sono quasi tecnicamente equivalenti alle classi astratte, a parte il fatto che non possono dichiarare variabili d’istanza. Anche se le interfacce possono oramai implementare metodi astratti e concreti come le classi astratte, non bisognerebbe mai dimenticare la differenza che c’è a livello di scopo tra i due concetti (cfr. paragrafo 7.7.6). Un’interfaccia dovrebbe rappresentare un comportamento che più classi potrebbero assumere, e un comportamento non si può istanziare. Una classe astratta dovrebbe invece rappresentare una generalizzazione troppo astratta per essere istanziata. Inoltre bisogna aver presente che una classe può estendere una sola classe astratta, mentre può implementare tante interfacce.

Per definizione prendono il nome di **interfacce funzionali** (in inglese **functional interfaces)** le interfacce che contengono un unico metodo astratto. Per indicare quest’ultimo viene spesso usato l’acronimo **SAM**, che sta per **Single Abstract Method** (in italiano **singolo metodo astratto).** Vengono

chiamate così proprio perché è come se esistessero solo per dichiarare una funzione da implementare. Le interfacce funzionali sono un ottimo esempio di come le interfacce dovrebbero essere utilizzate. Esiste anche un nuovo package che dichiara una serie di interfacce funzionali: il package `java.lang.function`. Anche sulle interfacce funzionali torneremo più avanti per spiegare le loro applicazioni, in particolare nel capitolo 16 dedicato alle espressioni lambda.

7.7.5 Ereditarietà multipla

Come abbiamo già detto, una classe può implementare più interfacce. Quindi in Java oggi esiste una versione semplificata di quella caratteristica chiamata *ereditarietà multipla*.

> **Parliamo di semplificazione perché le classi possono ereditare dalle interfacce solo la loro parte funzionale (i metodi) e non i dati (a parte le costanti statiche che un'interfaccia può dichiarare). In altri linguaggi invece esiste l'ereditarietà multipla completa, che comprende regole molto complicate per gestire i problemi derivanti dalla sua implementazione.**

Era possibile implementare più interfacce anche prima di Java 8 ma si ereditavano metodi astratti che bisognava riscrivere. Era tutto molto semplice anche nel caso si ereditassero due metodi con lo stesso nome da interfacce diverse: era sempre necessario riscrivere il metodo. Con l'avvento dei metodi di default l'ereditarietà multipla ha quindi assunto un significato diverso rispetto al passato. Quindi se consideriamo le seguenti interfacce:

```
public interface Lettore {
    default void leggi(Libro libro) {
        System.out.println("Sto leggendo: " +
libro.getTitolo()
          + " di " + libro.getAutore());
    }
}
public interface Programmatore {
```

```
    default void programma(String linguaggio) {
        System.out.println("Sto programmando in " +
linguaggio);
    }
}
```

è possibile creare la seguente classe che implementa entrambe le interfacce e ne eredita i metodi:

```
public class ChiStaLeggendo implements Lettore,
Programmatore {

}
```

Ed eccone un esempio di utilizzo:

```
public class TestEreditarietaMultipla {
    public static void main(String args[]) {
        ChiStaLeggendo tu = new ChiStaLeggendo();
        Libro java8 = new Libro("Manuale di Java 8","Claudio
De Sio Cesari");
        tu.programma("Java");
        tu.leggi(java8);
    }
}
```

E sin qui nessun problema. Ma non sarà sempre così semplice implementare l'ereditarietà multipla. Esistono una serie di casi in cui bisogna conoscere le regole per risolvere i problemi che si presentano. Vediamo le varie regole iniziando dal caso *Diamond Problem*.

7.7.5.1 Diamond Problem

Nel caso una classe erediti da due interfacce due metodi di default con la stessa firma (nome più lista di attributi) bisogna che la classe ridefinisca il metodo. Consideriamo per esempio la seguente interfaccia che dichiara un metodo di default:

```
public interface Solista {
    default void eseguiAssolo() {
        //Scala maggiore in DO
        System.out.println("DO RE MI FA SOL LA SI");
    }
```

```
}
```

Estendiamo questa interfaccia due volte con delle specializzazioni che riscrivono il metodo di default:

```
public interface SolistaBlues extends Solista {
    default void eseguiAssolo() {
        //Scala blues in DO
        System.out.println("DO MIb FA SOLb SOL SIb DO");
    }
}
public interface SolistaRock extends Solista {
    default void eseguiAssolo() {
        //Scala pentatonica in DO
        System.out.println("DO RE MI SOL LA DO");
    }
}
```

Se provassimo ora a creare una classe che implementi entrambe le interfacce, per esempio:

```
public class Chitarrista implements SolistaBlues,
SolistaRock {

}
```

otterremo il seguente errore in compilazione:

```
Chitarrista.java:1:
error: class Chitarrista inherits unrelated
  defaults for esegu
iAssolo() from types SolistaBlues and SolistaRock
public class Chitarr
ista implements SolistaBlues, SolistaRock {
       ^
1 error
```

Si tratta del famoso *problema dell'ereditarietà a diamante* (in inglese *Diamond Problem)*. In realtà dovrebbe essere tradotto come *problema dell'ereditarietà a rombo*. In questo caso infatti, il termine *diamond* dovrebbe essere tradotto come *rombo*, ma è talmente diffusa la traduzione (errata in questo caso) in *diamante* che oramai è uno standard trovare tale nomenclatura. Il termine *rombo* deriva dalla forma che viene fuori dalla

rappresentazione grafica dell'ereditarietà multipla, come si può vedere in figura 7.5.

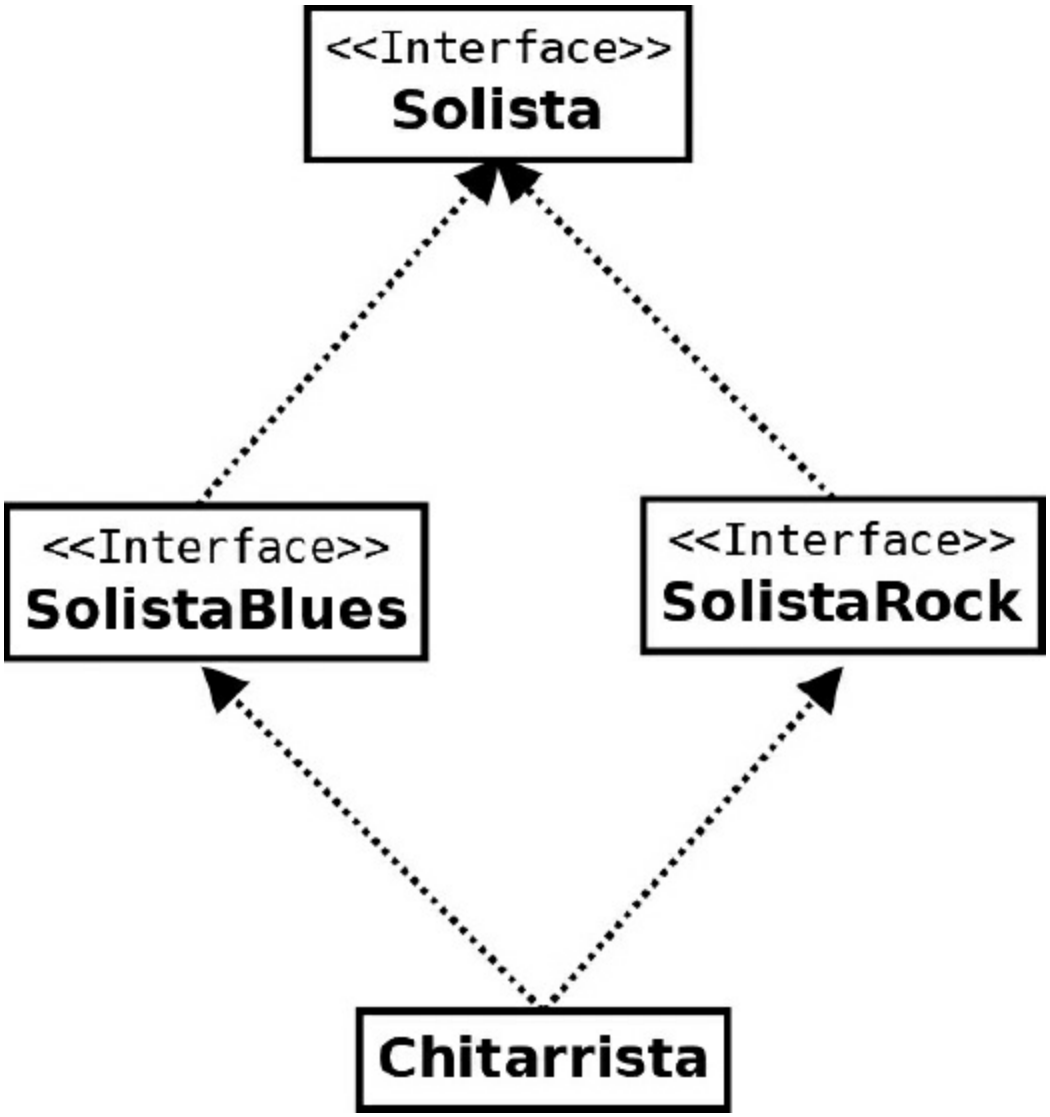


Figura 7.5 - Diamond Problem descritto con UML.

Nell'esempio il compilatore ci ha segnalato un errore, in quanto non è in grado di decidere quale delle due implementazioni ereditate del metodo `eseguiAssolo()` debba essere considerata quella prioritaria.

> **Notare che anche se non esistesse l'interfaccia `Solista`, il problema si presenterebbe esattamente allo stesso modo. Diciamo però che per ritrovarci in questa situazione è più**

probabile che l'interfaccia `Solista` sia stata definita.

L'unica soluzione che abbiamo per risolvere il conflitto di nomi è quello di ridefinire il metodo per specificarne l'implementazione nuovamente. Per esempio nella seguente classe creiamo un metodo completamente nuovo che usa le implementazioni di entrambe le interfacce implementate:

```
public class Chitarrista implements SolistaBlues,
SolistaRock {
    public void eseguiAssolo() {
        //Scala pentatonica + scala blues in DO
        System.out.println("DO RE MI SOL LA DO");
        System.out.println("DO MIb FA SOLb SOL SIb DO");
    }
}
```

L'output di questa classe sarà:

```
DO RE MI SOL LA DO
DO MIb FA SOLb SOL SIb DO
```

In questo caso abbiamo duplicato codice già scritto nelle interfacce, ma questo è evitabile utilizzando il reference `super`. Con la seguente sintassi infatti si ottiene lo stesso risultato:

```
public class Chitarrista implements SolistaBlues,
SolistaRock {
    public void eseguiAssolo() {
        //Scala pentatonica + scala blues in DO
        SolistaRock.super.eseguiAssolo();
        SolistaBlues.super.eseguiAssolo();
    }
}
```

La sintassi di `super` in questo caso è diversa da quella che abbiamo visto nel paragrafo 7.5.3. Infatti la semplice chiamata del tipo:

```
super.eseguiAssolo();
```

non avrebbe risolto l'ambiguità. Quindi si usa la sintassi del tipo:

```
NomeSuperClasse.super.nomeMetodoDellaSuperclasse
```

Chiarita la prima regola, ci sono altre possibili situazioni che bisogna gestire quando si implementa l'ereditarietà multipla.

7.7.5.2 Conflitto tra un metodo astratto e uno concreto

Nel caso una classe implementi più interfacce, se si presenta all'interno di essa un conflitto di nomi tra un metodo astratto ereditato dalla prima interfaccia, ed un metodo concreto ereditato da un'altra interfaccia, dovremo risolvere comunque la situazione ridefinendo il metodo nella classe così come visto nel caso *Diamond Problem*. Nonostante quindi sarebbe stato plausibile ereditare solo il metodo concreto, i progettisti Java hanno optato per l'uniformità. Non interessa come due interfacce paritetiche vanno in conflitto. Se almeno un'interfaccia fornisce un'implementazione al metodo in conflitto, il programmatore deve risolvere l'ambiguità con consapevolezza. Quindi se nell'esempio precedente l'interfaccia `SolistaRock` avesse definito come astratto il metodo `eseguiAssolo()`, allora la classe Chitarrista avrebbe comunque dovuto riscrivere il metodo come nel caso *Diamond Problem*.

7.7.5.3 Conflitto tra due metodi astratti

In una classe che implementa più interfacce, se c'è conflitto di nomi di soli metodi astratti, allora ci troviamo nella situazione pre-Java 8, e la soluzione è evidentemente quella di ridefinire il metodo nella sotto classe (così come visto per la soluzione al *Diamond Problem)*.

Ovvero, se nell'esempio precedente entrambe le interfacce `SolistaRock` e `SolistaBlues` avessero definito come astratti i metodi `eseguiAssolo()`, allora la classe `Chitarrista` avrebbe ereditato un metodo astratto che andava riscritto. Quindi la soluzione descritta per il *Diamond Problem* è valida anche in questo caso.

7.7.5.4 Conflitto tra metodi ridefiniti

Nel caso una classe implementi due interfacce dove una estende l'altra, se nella classe che le implementa c'è conflitto di nomi di metodi ereditati, allora verrà ereditata l'implementazione della interfaccia più specifica.

Quindi, considerando le interfacce definite nel caso *Diamond Problem*, se una nuova classe `ChitarristaRock` avesse implementato le interfacce

`Solista` e `SolistaRock` come nel seguente esempio:

```
public class ChitarristaRock implements Solista, SolistaRock
{

}
```

allora la classe `ChitarristaRock` avrebbe ereditato l'implementazione dell'interfaccia `SolistaRock`, in quanto più specifica rispetto a quella di `Solista` (infatti `SolistaRock` aveva ridefinito il metodo di `Solista`). Quindi l'output della seguente classe di test:

```
public class Concerto {
    public static void main(String args[]) {
        Chitarrista chitarrista = new Chitarrista();
        chitarrista.eseguiAssolo();
        ChitarristaRock chitarristaRock = new
ChitarristaRock();
        chitarristaRock.eseguiAssolo();
    }
}
```

sarebbe il seguente:

```
DO RE MI SOL LA DO
DO MIb FA SOLb SOL SIb DO
DO RE MI SOL LA DO
```

7.7.5.5 Class always wins

Nel caso una classe erediti un conflitto di firme tra un metodo ereditato da una classe estesa e un metodo dell'interfaccia implementata, allora verrà ereditata l'implementazione della classe, anche se questa fosse astratta (questa regola è nota con il nome di **classe vince sempre** in inglese **class always wins).** Per esempio consideriamo la seguente classe astratta:

```
public abstract class ChitarristaAstratto {
    public abstract void eseguiAssolo();
}
```

che definisce il metodo `eseguiAssolo()` in maniera astratta. Se provassimo ora a compilare la seguente classe `ChitarristaSolista` che estende la classe

`ChitarristaAstratto` e implementa l'interfaccia `Solista` (che invece definisce un'implementazione di default al metodo `eseguiAssolo()`):

```
public class ChitarristaSolista extends ChitarristaAstratto
  implements Solista {
}
```

avremmo il seguente errore in compilazione:

```
ChitarristaSolista.java:1: error: ChitarristaSolista is not
abstract and
  does not override abstract method eseguiAssolo() in
ChitarristaAstratto
public class ChitarristaSolista extends ChitarristaAstratto
  implements Solista {
       ^
1 error
```

Infatti, tra le due definizioni del metodo `eseguiAssolo()`, è stata ereditata la versione (benché astratta) della classe `ChitarristaAstratto` piuttosto che la versione concreta dell'interfaccia `Solista` (class always wins). Se però la classe `ChitarristaAstratto` avesse implementato in qualche modo il metodo `eseguiAssolo()`, allora la classe `ChitarristaSolista` sarebbe stata compilata senza errori, ereditando il metodo `eseguiAssolo()` della classe `ChitarristaAstratto`.

Riassumendo le cinque regole appena viste: il metodo ereditato si deve sempre riscrivere tranne negli ultimi due casi (*Conflitto tra metodi ridefiniti* e *Class always wins)* dove vincono rispettivamente l'interfaccia più specifica e la classe.

7.7.6 Differenze tra interfacce e classi astratte

Non è possibile istanziare né classi astratte né interfacce. Inoltre il vantaggio comune che offrono sia le classi astratte sia le interfacce risiede nel fatto che esse possono obbligare le sottoclassi ad implementare comportamenti. Una classe che eredita un metodo astratto, infatti, deve fare override del metodo ereditato oppure essere dichiarata a sua volta astratta. Dal punto di vista della progettazione, quindi, questi strumenti supportano l'astrazione dei dati. Un'evidente differenza pratica tra i due concetti è cosa possono definire.

Un’interfaccia può definire costanti statiche, metodi statici, metodi di default, metodi ausiliari privati (anche statici) e metodi astratti. Una classe astratta è invece una classe normale, che però non può essere istanziata e che può contenere metodi astratti (ma questo non è necessario).

Dunque una classe astratta solitamente non è altro che un’astrazione troppo generica per essere istanziata nel contesto in cui si dichiara. Un buon esempio è la classe `Veicolo`.

```
public abstract class Veicolo {
    public abstract void accelera();
    public abstract void decelera();
}
```

Per esempio, ogni veicolo accelera e decelera e quindi tutte le classi non astratte che estendono `Veicolo` devono riscrivere i metodi `accelera()` e `decelera()`.

```
public class Aereo extends Veicolo {
    public void accelera() {
        // override del metodo ereditato
        //...
    }
    public void decelera() {
        // override del metodo ereditato
        //...
    }
    //...
}
```

Un’interfaccia invece, di solito non è una vera astrazione troppo generica per il contesto, semmai è una *astrazione comportamentale*, che non ha senso istanziare nel contesto stesso. Spesso le interfacce hanno nomi che richiamano aggettivi e comportamenti. Considerando sempre l’esempio del `Veicolo` superclasse di `Aereo`, potremmo introdurre un’interfaccia `Volante` (notare come il nome faccia pensare ad un comportamento più che ad un oggetto astratto) che sarà implementata dalle classi che volano. Ora, se l’interfaccia `Volante` è definita nel seguente modo:

```
public interface Volante {
    void atterra();
    void decolla();
```

```
}
```

ogni classe che deve astrarre un concetto di oggetto volante (come un aereo o un uccello) deve implementare l'interfaccia. Riscriviamo quindi la classe `Aereo` nel seguente modo:

```
public class Aereo extends Veicolo implements Volante {
    public void atterra() {
        // override del metodo di Volante
    }
    public void decolla() {
        // override del metodo di Volante
    }
    public void accelera() {
        // override del metodo di Veicolo
    }
    public void decelera() {
        // override del metodo di Veicolo
    }
}
```

Il vantaggio di programmare con classi astratte ed interfacce sarà spiegato nel prossimo capitolo dedicato al polimorfismo.

Riepilogo

Questo capitolo è stato dedicato principalmente al secondo paradigma fondamentale dell'Object Orientation: l'**ereditarietà.** Si applica tramite la parola chiave **`extends`**, con la sottoclasse che deve estendere la superclasse. È un concetto chiaro e semplice da apprendere e l'applicabilità si basa sulla relazione **"is a".** Per esempio considerando le classi `Trapezio` e `Triangolo`, una volta che ci siamo accorti che hanno delle caratteristiche in comune dobbiamo chiederci: "ma un trapezio è un triangolo?". Ovviamente no, ma come abbiamo visto sia un trapezio sia un triangolo sono poligoni, e quindi è opportuno creare la superclasse `Poligono`. Ovvero, invece di utilizzare il processo della **specializzazione** che porta a creare da una superclasse una sottoclasse, in questo caso bisogna utilizzare il processo di **generalizzazione**, che consiste nel creare una superclasse che raccolga gli elementi comuni a più sottoclassi.

Sono stati anche introdotti vari altri argomenti correlati con l'ereditarietà.

Il **modificatore `final`** impedisce l'applicazione dell'estensione alle classi o la riscrittura dei metodi ereditati.

Il **modificatore `abstract`** è usato per lo scopo opposto, ovvero costringere le sottoclassi ad estendere le classi astratte e a riscrivere i metodi astratti. Una classe astratta può avere metodi astratti ma non è obbligatorio che ne abbia almeno uno. I metodi astratti non hanno un blocco di codice, e vengono definiti per essere implementati nelle sottoclassi. La **classe `Object`** è la superclasse di qualsiasi classe, diretta o indiretta. Infatti una qualsiasi classe che non estende un'altra classe, estende implicitamente `Object`.

È stato evidenziato che **l'ereditarietà e l'incapsulamento coesistono tranquillamente**, anzi si convalidano a vicenda. Le variabili private di una superclasse vengono incapsulate nella superclasse, e quindi possono essere utilizzate anche nella sottoclasse sfruttando i metodi setter e getter. Inoltre l'utilizzo corretto dell'astrazione è la base per non commettere errori che sarebbero amplificati da incapsulamento ed ereditarietà.

I **costruttori non si ereditano**, ma la **prima istruzione in un costruttore**, è sempre una chiamata al costruttore della superclasse. La sintassi **`super()`** viene inserita dal compilatore nel momento in cui è assente da un costruttore. Di conseguenza i costruttori di una superclasse sono sempre chiamati dai costruttori di una sottoclasse. La chiamata ad un altro costruttore della stessa classe tramite il comando `this()`, implica solo che la chiamata a `super()` sia rimandata. Infatti i costruttori di una stessa classe non possono richiamarsi tutti tra loro, ed almeno uno non avrà come prima istruzione una chiamata di tipo `this()`. Quindi, esplicitamente o implicitamente in almeno un costruttore avremo una chiamata di tipo `super()`.

I metodi statici sono ereditati a livello di classe, creando una sorta di **ereditarietà di classe.**

Il **modificatore `private`** applicato ai membri di una classe fa in modo che essi non siano ereditati. Il **modificatore `protected`** invece, fa ereditare i membri alle sottoclassi, ma non è necessario per le variabili se esse sono incapsulate, anche perché le variabili protette sono visibili come fossero pubbliche alle classi del package in cui sono dichiarate. **Non usare modificatori** rende i membri di una classe visibili solo all'interno dello stesso package, e quindi non saranno ereditati da classi esterne a tale package.

Se una classe non è dichiarata `public`, e quindi non dichiara modificatori, essa non è visibile all'esterno del package in cui è dichiarata. Questo implica che non può essere estesa da classi non appartenenti allo stesso package.

Gli **inizializzatori** sono eseguiti quando viene caricato in memoria un oggetto, mentre gli inizializzatori statici sono caricati quando una classe viene caricata in memoria. Siccome le sottoclassi dipendono dalle loro superclassi, allora gli inizializzatori di una superclasse, anche se non sono ereditati, vengono **comunque invocati** per essere caricati in memoria.

Le **interfacce dovrebbero** essere create per rappresentare comportamenti. Non possono dichiarare variabili d'istanza ma solo costanti statiche e pubbliche. Possono dichiarare metodi astratti, metodi di default, metodi statici. Dalla versione 9 di Java possono anche dichiarare metodi ausiliari privati, sia statici che non. I metodi privati non possono essere dichiarati come metodi di default.

È possibile implementare una sorta di **ereditarietà multipla** in Java, implementando più interfacce che hanno metodi di default. È possibile incappare nel famoso **problema dell'ereditarietà a diamante**, che si presenta quando viene ereditato lo stesso metodo da due interfacce diverse. Le regole in tali casi si possono riassumere nel fatto che il metodo ereditato si deve sempre riscrivere tranne che in due casi. Nel caso le due interfacce da cui viene ereditato il metodo sono una superclasse dell'altra, allora viene preso in considerazione direttamente il metodo della sottoclasse, perché più specifico. Un altro caso dove non c'è bisogno di riscrivere il metodo, è quando ereditiamo lo stesso metodo da una classe e da un'interfaccia: "vince sempre il metodo della classe".

La **differenza essenziale** tra le classi astratte e le interfacce risiede nel fatto che le classi astratte dovrebbero rappresentare concetti troppo generici per essere istanziati. Mentre le interfacce dovrebbero rappresentare comportamenti. Un buon esempio di utilizzo delle interfacce, sono le **interfacce funzionali**, ovvero interfacce che dichiarano un unico metodo astratto (detto **SAM**, Single Abstract Method).

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere la definizione, l'utilizzo e l'utilità dell'ereditarietà (generalizzazione e specializzazione) (unità 7.1, 7.5)	☐	
Conoscere le conseguenze dell'utilizzo contemporaneo di incapsulamento ed ereditarietà (unità 7.4)	☐	
Comprendere e saper utilizzare i modificatori `abstract` e `final` (unità 7.3, 7.6)	☐	
Comprendere e saper utilizzare la parola chiave `super` (unità 7.5)	☐	
Comprendere e saper le interfacce (unità 7.7)	☐	
Sapersi destreggiare nei vari casi complessi dell'ereditarietà multipla (unità 7.7)	☐	

Note:

8

Polimorfismo

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Comprendere il significato del polimorfismo (unità 8.1).
- ✔ Saper utilizzare l'overload, l'override ed il polimorfismo per metodi (unità 8.2 e 8.3).
- ✔ Comprendere e saper utilizzare il polimorfismo per dati con le sue implementazioni: le collezioni eterogenee, i parametri polimorfi e le invocazioni virtuali di metodi (unità 8.3).
- ✔ Sapere utilizzare l'operatore `instanceof` ed il casting di oggetti (unità 8.3).

Questo capitolo è interamente dedicato al paradigma più complesso dell'Object Orientation: il **polimorfismo.** Si tratta di un argomento abbastanza vasto ed articolato che viene sfruttato relativamente poco rispetto alla potenza che mette a disposizione dello sviluppatore. Molti programmatori Java non riescono neanche a definire questo fondamentale strumento di programmazione. Alla fine di questo capitolo, invece, il lettore dovrebbe comprenderne pienamente l'importanza. Nel caso non fosse così, potrà sempre tornare a leggere queste pagine in futuro. Una cosa è certa: comprendere il polimorfismo è molto più semplice che implementarlo.

8.1 Polimorfismo

Il **polimorfismo** (dal greco “molte forme”) è un altro concetto che dalla realtà è stato importato nella programmazione ad oggetti. Esso permette di riferirci con un unico termine a entità diverse. Ad esempio, sia un telefono fisso sia uno smartphone consentono di telefonare, dato che entrambi i mezzi sono definibili come telefoni. Telefonare, quindi, può essere considerata un’azione polimorfica (ha diverse implementazioni). Il polimorfismo in Java è argomento complesso, che si dirama in vari sottoargomenti. Utilizzando una convenzione con la quale rappresenteremo mediante rettangoli i concetti, e con ovali i concetti che hanno una reale implementazione in Java, cercheremo di schematizzare il polimorfismo e le sue espressioni.

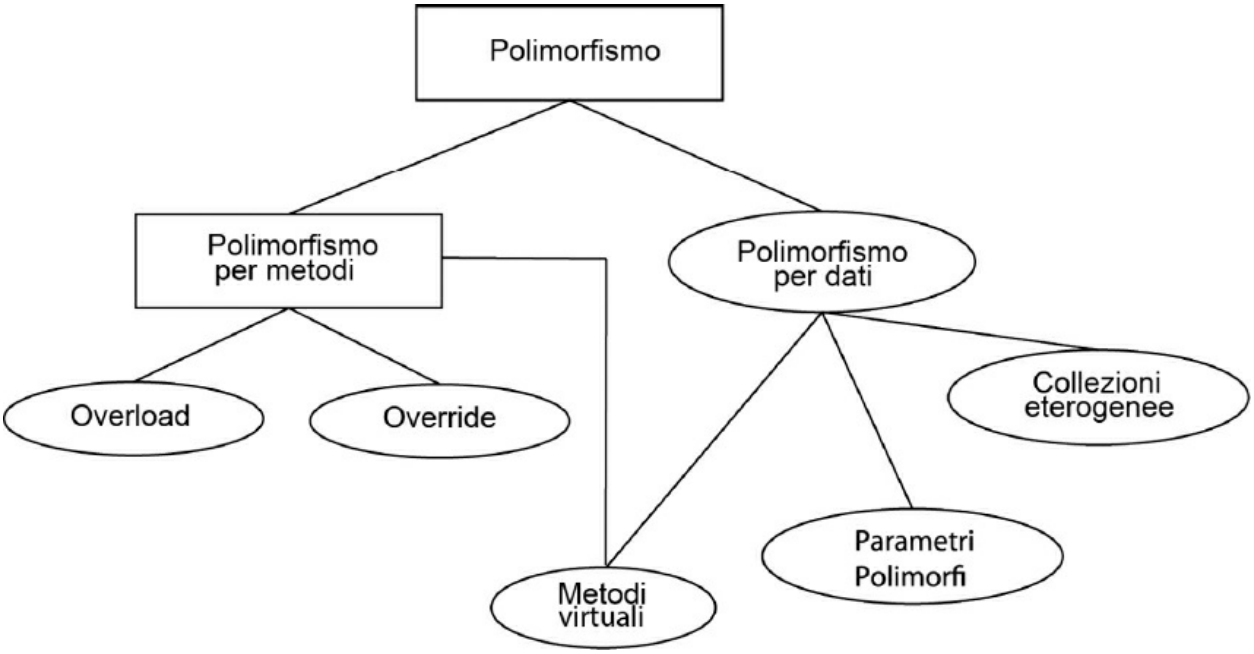


Figura 8.1 - Il polimorfismo in Java.

8.1.1 Convenzione per i reference

Prima di iniziare a definire i vari aspetti del polimorfismo presentiamo una convenzione per la definizione di una variabile di tipo reference. Definire precisamente che cosa sia un reference e come sia rappresentato in memoria non è cosa semplice. Di solito s’identifica un reference con un puntatore (anche se non è propriamente corretto). In molti testi relativi ad altri linguaggi di programmazione, un puntatore viene definito come “una

variabile che contiene un indirizzo". In realtà la definizione di puntatore cambia da piattaforma a piattaforma! Quindi ancora una volta utilizziamo una convenzione. Possiamo definire un **reference** come una variabile che contiene due informazioni rilevanti: l'indirizzo in memoria e l'intervallo di puntamento definito dalla relativa classe. Consideriamo la classe `Punto`: che abbiamo già incontrato, modificata con l'incapsulamento.

```
public class Punto {
    private int x;
    private int y;

    public void setX(int x) {
        this.x = x;
    }
    public void setY(int y) {
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

Per esempio, se scriviamo:

```
Punto ogg = new Punto();
```

possiamo supporre che il reference `ogg` abbia come indirizzo un valore numerico, ad esempio `10023823`, e come intervallo di puntamento `Punto`. In particolare, ciò che abbiamo definito come intervallo di puntamento farà sì che il reference `ogg` possa accedere all'interfaccia pubblica della classe `Punto`, ovvero a tutti i membri pubblici (`setX`(), `setY()`, `getX()`, `getY()`) dichiarati nella classe `Punto` tramite il reference `ogg`. L'indirizzo invece farà puntare il reference ad una particolare area di memoria dove risiederà il particolare oggetto istanziato. Viene riportato uno schema rappresentativo in figura 8.2.

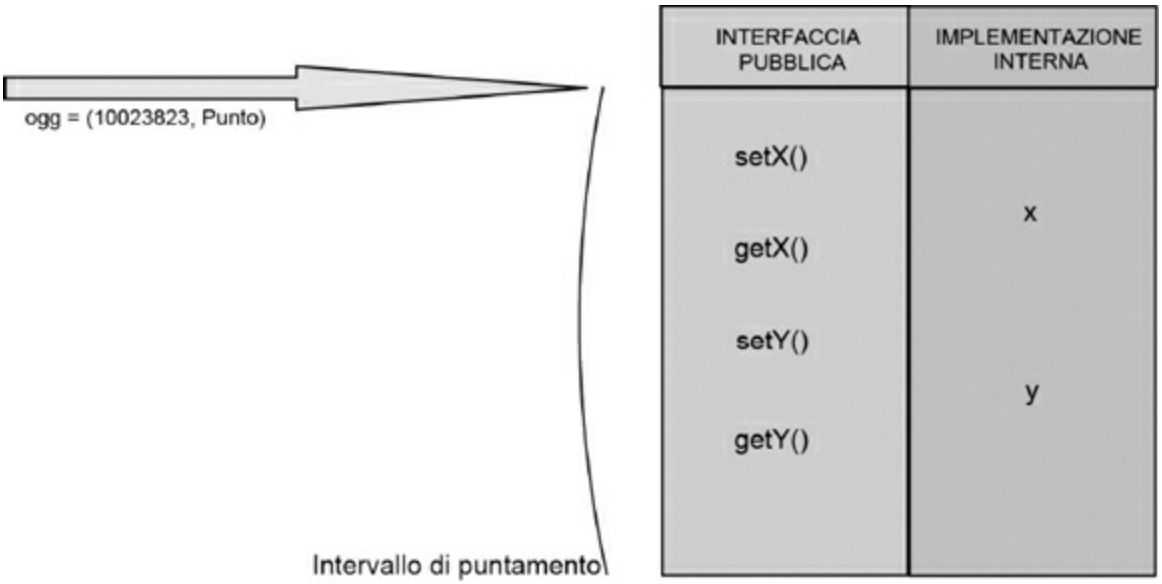


Figura 8.2 - Convenzione per i reference.

8.2 Polimorfismo per metodi

Il **polimorfismo per metodi**, per quanto asserito sinora, ci permetterà di utilizzare lo stesso nome per metodi differenti. In Java esso trova una sua realizzazione pratica sotto due forme: l'**overload** (che potremmo tradurre con "sovraccarico") e l'**override** (che potremmo tradurre con "riscrittura").

8.2.1 Overload

In un metodo, la coppia costituita dall'identificatore e dalla lista dei parametri è detta **firma** o **segnatura** (dall'inglese **signature)** del metodo.

In Java un metodo è univocamente determinato non solo dal suo identificatore ma anche dalla sua lista di parametri, cioè dalla sua firma. Quindi, in una classe possono convivere metodi con lo stesso nome ma con differente firma. Su questo semplice concetto si fonda una delle implementazioni più utilizzate di Java: l'**overload.** Mediante questa pratica il programmatore potrà utilizzare lo stesso nome per metodi diversi. Tutto ciò dovrà comunque avere un senso logico. Per esempio potremmo assegnare lo stesso nome a due metodi che concettualmente hanno la stessa funzionalità, ma soddisfano tale funzionalità in maniera differente. Presentiamo di seguito un banale esempio di overload:

```
public class Aritmetica {
    public int somma(int a, int b) {
        return a + b;
    }
    public float somma(int a, float b) {
    return a + b;
    }
    public float somma(float a, int b) {
        return a + b;
    }
    public int somma(int a, int b, int c) {
        return a + b + c;
    }
    public double somma(int a, double b, float c) {
        return a + b + c;
    }
}
```

In questa classe ci sono ben cinque metodi che hanno lo stesso nome ed eseguono somme, ma in modo differente. Se volessimo implementare questi metodi in un altro linguaggio che non supporta l'overload, dovremmo inventare un nome nuovo per ogni metodo. Per esempio il primo di essi si potrebbe chiamare `sommaDueInt()`, il secondo `sommaUnIntEUnFloat()`, il terzo `sommaUnFloatEUnInt()`, il quarto `sommaTreInt()`, il quinto addirittura `sommaUnIntUnDoubleEUnFloat()`! A questo punto l'utilità dell'overload è evidente.

Notiamo che la lista dei parametri ha tre criteri di distinzione:

- **di tipo** (es.: `somma(int a, int b)` è diverso da `somma(int a, float b))`
- **numerico** (es.: `somma(int a, int b)` è diverso da `somma(int a, int b, int c))`
- **posizionale** (es.: `somma(int a, float b)` è diverso da `somma(float a, int b))`

Gli identificatori che utilizziamo per i parametri invece, non sono criteri di distinzione per i metodi (esempio: `somma(int a, int b)` non è diverso da `somma(int c, int d))`.

Il tipo di ritorno inoltre, non fa parte della firma di un metodo, quindi non ha importanza per l'implementazione dell'overload.

In alcuni testi l'overload non è considerato aspetto polimorfico di un linguaggio. In questi testi il polimorfismo stesso è definito in maniera diversa da com'è stato definito in questo contesto. Come sempre, è tutto relativo all'ambito in cui ci si trova. Se ci fossimo trovati a discutere di analisi e progettazione object oriented anziché di programmazione, neanche noi avremmo inserito l'overload come argomento del polimorfismo. Se prescindiamo dal linguaggio infatti, ma non è il nostro caso, l'overload non dovrebbe neanche esistere. Inoltre il polimorfismo è stato definito da molti autori come una conseguenza all'implementazione dell'ereditarietà. L'overload e l'ereditarietà in effetti non hanno nulla in comune. Se ci limitiamo però a considerare la definizione che abbiamo dato di polimorfismo (stesso nome a cose diverse), l'overload è sicuramente da considerarsi una delle implementazioni del polimorfismo. Qualche autore invece definisce l'overload come "polimorfismo statico" per differenziarlo dall'override (sovrascrittura di metodi nelle sottoclassi) che viene etichettato come "polimorfismo dinamico".

Altri esempi di overload che abbiamo già sfruttato più o meno inconsapevolmente in questo testo, sono i metodi `println()` e `print()` dell'oggetto `System.out`. Infatti esistono ben dieci metodi `println()` diversi, e nove versioni di `print()`. È possibile passare a questi metodi non solo stringhe, ma anche interi, booleani, array di caratteri o addirittura `Object`. Il lettore può verificarlo per esercizio consultando la documentazione (suggerimento: `System.out` è un oggetto di tipo `PrintStream` appartenente al package `java.io`).

Avevamo anche visto un altro esempio di overload già nel capitolo 2, quando abbiamo introdotto i costruttori, ed anche nel capitolo 6 dove abbiamo visto come un costruttore può invocare un altro costruttore mediante la parola chiave `this`. Ci era parso naturale creare più costruttori in una stessa classe con differenti liste di parametri, in realtà stavamo già utilizzando

l’overload inconsapevolmente. Segue un semplice esempio:

```
public class Cliente {
    private String nome;
    private String indirizzo;
    private String numeroDiTelefono;
    public Cliente() {
        // costruttore inserito esplicitamente (non di
default)
    }
    public Cliente(String nome) {
        this.nome = nome;
    }
    public Cliente(String nome, String indirizzo) {
        this(nome);
        this.indirizzo = indirizzo;
    }
    public Cliente(String nome, String indirizzo, String
numeroDiTelefono) {
        this(nome, indirizzo);
        this.numeroDiTelefono = numeroDiTelefono;
    }
//...
}
```

8.2.2 Varargs

Come già accennato quando abbiamo presentato il concetto di metodo nel capitolo 2, è possibile utilizzare come argomenti dei metodi i cosiddetti **varargs** (abbreviativo di **variable arguments).** Con i varargs è possibile fare in modo che un metodo accetti un numero non precisato di argomenti (compresa la possibilità di passare zero parametri), evitando così un eccessivo overload di metodi. La sintassi dei varargs fa uso di **ellissi** (i puntini sospensivi “...”). Per esempio, la seguente implementazione della classe `Aritmetica` potrebbe sostituire con un unico metodo l’overload dell’esempio del paragrafo precedente:

```
public class Aritmetica {
    public double somma(double... doubles) {
        double risultato = 0.0D;
        for (double tmp :  doubles) {
            risultato += tmp;
```

```
        }
        return risultato;
    }
}
```

Infatti, tenendo conto che `ogg` è un oggetto di tipo `Aritmetica`, il seguente codice è valido:

```
System.out.println(ogg.somma(1,2,3));
System.out.println(ogg.somma());
System.out.println(ogg.somma(1,2));
System.out.println(ogg.somma(1,2,3,5,6,8,2,43,4));
```

Segue l'output delle precedenti righe di codice:

```
6.0
0.0
3.0
74.0
```

Il risultato sarà di tipo `double` (a meno di casting).

Effettivamente i varargs, all'interno del metodo dove sono dichiarati, sono considerati a tutti gli effetti degli array. Quindi, come per gli array, se ne può ricavare la dimensione con la variabile `length` e usarli nei cicli. Il vantaggio di avere varargs in luogo di un array o di una collection risiede essenzialmente nel fatto che per chiamare un metodo che dichiara argomenti variabili, non bisogna creare array o collection.

Con il termine "collection", intendiamo un insieme di classi ed interfacce (chiamato "framework collections") definito dalla libreria standard, che rappresentano collezioni di oggetti. Rispetto agli array, le collection sono ridimensionabili e hanno caratteristiche ben precise. Per esempio le classi che implementano l'interfaccia `List` hanno la caratteristica di essere ordinate, e ad ogni elemento è associato un indice. Invece le classi che implementano l'interfaccia `Set` possiedono la caratteristica di non ammettere duplicati. La collection probabilmente più usata è l'`ArrayList`, che estende l'interfaccia `List`. Giusto per

dare un'idea di cosa stiamo parlando, di seguito mostriamo come si crea e si riempie un `ArrayList` di stringhe:

```
ArrayList arrayList = new ArrayList();
arrayList.add("ArrayList");
arrayList.add("implementa");
arrayList.add("List");
```

approfondiremo bene il discorso nel capitolo relativo alle librerie fondamentali.

Un metodo con varargs viene semplicemente invocato come si fa con un qualsiasi overload. Se per esempio avessimo avuto un array al posto di varargs per il metodo `somma()`:

```
public double somma(double[] doubles) {
    double risultato = 0.0D;
    for (double tmp :  doubles) {
        risultato += tmp;
    }
    return risultato;
}
```

per invocare il metodo avremmo dovuto ricorrere alle seguenti righe di codice:

```
double[] doubles = {1.2D, 2, 3.14, 100.0};
System.out.println(ogg.somma(doubles));
```

Insomma, usare i varargs in luogo di array come parametri di un metodo è più comodo.

I varargs danno l'illusione dell'utilizzo dell'overload anche se parliamo di qualcosa di diverso. Infatti i varargs vanno dichiarati specificando un certo tipo di dato (per esempio `double` o `String`) e il numero di valori da passare è indefinito (può essere anche zero). Con l'overload possiamo specificare un numero fisso di parametri e di tipi diversi. Inoltre, riguardo i varargs:

1. è possibile dichiarare un unico varargs a metodo;

2. è possibile dichiarare anche altri parametri oltre ad un (unico) varargs per metodo, ma il varargs deve occupare l'ultima posizione tra i parametri.

Per esempio consideriamo di nuovo la classe `Cliente` che definiva un overload di costruttori definito nel paragrafo 8.2.1. Con i varargs potremmo creare un unico costruttore in questo modo:

```
public class ClienteVarargs {
    private String nome;
    private String indirizzo;
    private String numeroDiTelefono;

    public ClienteVarargs(String... par) {
        if (par != null) {
            if (par[0] != null) {
                this.nome = par[0];
            }
            if (par[1] != null) {
                this.indirizzo = par[1];
            } if (par[2] != null) {
                this.numeroDiTelefono = par[2];
            }
        }
    }
//...
}
```

Sembra evidente che l'overload rappresenti una soluzione migliore in questo caso.

8.2.3 Override

L'**override**, e questa volta non ci sono dubbi, è invece considerato una importantissima caratteristica della programmazione ad oggetti, ed è da qualcuno superficialmente identificato con il polimorfismo stesso. L'override (che potremmo tradurre con "riscrittura") è il termine object oriented che viene utilizzato per descrivere la caratteristica che hanno le sottoclassi di ridefinire un metodo ereditato da una superclasse. Ovviamente non esisterà override senza ereditarietà. Una sottoclasse è sempre più specifica della classe che estende, e quindi potrebbe ereditare metodi che hanno bisogno di

essere ridefiniti per funzionare correttamente nel nuovo contesto.

Ad esempio, supponiamo che una ridefinizione della classe `Punto` (che per convenzione assumiamo bidimensionale) dichiari un metodo `distanzaDallOrigine()` che calcola la distanza tra un punto di determinate coordinate e l'origine degli assi cartesiani. Questo metodo se ereditato all'interno di un'eventuale sottoclasse `PuntoTridimensionale`, ha bisogno di essere ridefinito per calcolare la distanza voluta, tenendo conto anche della terza coordinata. Vediamo quanto appena descritto sotto forma di codice:

```
public class Punto {
    private int x, y;

    public void setX(int x) {
        this.x = x;
    }
    public int getX() {
        return x;
    }
    public void setY(int y) {
        this.y = y;
    }
    public int getY() {
        return y;
    }
    public double distanzaDallOrigine() {
        int tmp = (x*x) + (y*y);
        return Math.sqrt(tmp);
    }
}

public class PuntoTridimensionale extends Punto {
    private int z;

    public void setZ(int z) {
        this.z = z;
    }
    public int getZ() {
        return z;
    }

    public double distanzaDallOrigine() {
        int tmp = (getX()*getX()) + (getY()*getY())
            + (z*z); // N.B.: x e y non sono ereditate
```

```
            return Math.sqrt(tmp);
        }
    }
```

Per chi non ricorda come si calcola la distanza geometrica tra due punti, se abbiamo i punti `P1` e `P2` con coordinate rispettivamente (`x1, y1`) e (`x2, y2`), la distanza tra essi sarà data da:

$$d(P1,P2) = \sqrt{(x1-x2)^2+(y1-y2)^2}$$

Se `P2` coincide con l'origine, ha coordinate (`0,0`) e quindi la distanza tra un punto `P1` e l'origine sarà data da:

$$d(P1,P2) = \sqrt{(x1)^2+(y1)^2}$$

Il metodo `sqrt()` **della classe** `Math` **(package** `java.lang`**) restituisce un valore di tipo** `double`**, risultato della radice quadrata del parametro passato (il lettore è invitato sempre e comunque a consultare la documentazione). Ricordiamo che è stato possibile invocarlo con la sintassi** `NomeClasse.nomeMetodo` **anziché** `nomeOggetto.nomeMetodo` **perché trattasi di un metodo statico.**

Il quadrato di `x` e di `y` è stato ottenuto mediante la moltiplicazione del valore per sé stesso. Per esercizio il lettore può esplorare la documentazione della classe `Math` per cercare l'eventuale presenza di un metodo per elevare numeri a potenza (suggerimento: "potenza" in inglese si dice "power").

Attenzione! Astrarre un punto bidimensionale creando una classe di nome `Punto` **e inserire il metodo** `distanzaDallOrigine()` **nella stessa classe, sono scelte che sembrano accettabili. Ma non stiamo così violando il paradigma dell'astrazione? La classe non dovrebbe chiamarsi** `PuntoBidimensionale?` **È giusto che un punto**

possegga un metodo per calcolare la distanza da un altro punto? I punti non calcolano la distanza nel mondo reale, semmai potrebbe essere utile creare una classe `Righello!` **Il discorso continua nell'appendice on line G intitolata "Un esempio guidato alla programmazione ad oggetti".**

Osservare come abbiamo ridefinito il blocco di codice del metodo `distanzaDallOrigine()` nella sottoclasse `PuntoTridimensionale`. Abbiamo introdotto le terza coordinata di cui tener conto affinché il calcolo della distanza fosse eseguito correttamente.

Si noti che abbiamo avuto accesso diretto alla variabile z (senza passare per il metodo set o get), giacché essa è definita privata nella stessa classe del metodo che la sta usando.

L'override sembra semplice quindi da implementare, ma non è tutto così scontato:

ci sono **regole** da rispettare:

1. il metodo riscritto nella sottoclasse deve avere la stessa firma (nome e parametri) del metodo della superclasse;
2. il tipo di ritorno del metodo della sottoclasse deve coincidere con quello del metodo che si sta riscrivendo, o deve essere di un tipo che estende il tipo di ritorno del metodo della superclasse;
3. il metodo ridefinito nella sottoclasse non deve essere meno accessibile del metodo originale della superclasse.

Per quanto riguarda la **prima regola**, dando per scontato che il nome del metodo della sottoclasse deve essere identico a quello della superclasse, se parliamo di firme differenti, parliamo di lista di parametri differenti. In casi come questo saremmo di fronte ad un overload in luogo di un override. Infatti non essendo riscritto con la stessa firma, erediteremmo nella sottoclasse

anche il metodo della superclasse, e ci sarebbero due metodi con lo stesso nome e differente lista di parametri. Per esempio, supponiamo che la classe `Punto` definisca il metodo `setCoordinate()`:

```
public void setCoordinate(int x, int y) {
    setX(x);
    setY(y);
}
```

Nella sottoclasse `PuntoTridimensionale` scriveremmo probabilmente:

```
public void setCoordinate(int x, int y, int z) {
    super.setCoordinate(x, y);
    setZ(z);
}
```

Ma questo metodo ha differente lista di parametri rispetto a quello definito in `Punto`, e quindi non è lo stesso metodo, non è un override. Di conseguenza nella sottoclasse `PuntoTridimensionale` avremo ereditato anche il metodo `setCoordinate()` della classe `Punto`.

Con la **seconda regola** risulta invece chiaro che anche il tipo di ritorno è una discriminante (a differenza dell'overload dove invece non lo era). In particolare il tipo di ritorno del metodo può coincidere anche con una sottoclasse del tipo di ritorno del metodo originale. In questo caso si parla di **tipo di ritorno covariante.** Per esempio, sempre considerando il rapporto di ereditarietà che sussiste tra le classi `Punto` e `PuntoTridimensionale`, se nella classe `Punto`, fosse presente il seguente metodo:

```
public Punto elaboraPunto() {
    //...
}
```

allora sarebbe legale implementare il seguente override nella classe `PuntoTridimensionale`:

```
public PuntoTridimensionale elaboraPunto() {
    //...
}
```

Infine per la **terza regola** la situazione è abbastanza chiara. Per esempio, se un metodo ereditato è dichiarato protetto, non si può ridefinire privato,

semmai pubblico.

> **Esiste anche un'altra regola che sarà trattata nel capitolo 9 relativo alle eccezioni. Inoltre nel capitolo 11 vedremo come implementare una sorta di "parametri covarianti" di un metodo, con una strategia basata sui tipi generici.**

8.2.4 Annotazione sull'override

Uno degli errori più subdoli che può commettere un programmatore è sbagliare un override. Ovvero ridefinire in maniera non corretta un metodo che si vuole riscrivere in una sottoclasse, magari digitando una lettera minuscola piuttosto che una maiuscola, o sbagliando il numero di parametri in input. In tali casi il compilatore non segnalerà errori, dal momento che non può intuire il tentativo di override in atto. Quindi, per esempio, il seguente codice:

```
public class PuntoTridimensionale extends Punto {
    public double distanzadallOrigine() { // si scrive
distanzaDallOrigine()!
        //...
    }
}
```

è compilato correttamente, ma non sussisterà nessun override in `PuntoTridimensionale`. Il guaio è che il problema si presenterà solo in fase di esecuzione dell'applicazione e non è detto che lo si riesca a correggere velocemente. Abbiamo accennato nel capitolo 2 al concetto di annotazione. Un**'annotazione** consente di *annotare* qualsiasi tipo di componente di un programma Java: dalle variabili ai metodi, dalle classi alle annotazioni stesse. Con il termine *annotare* intendiamo qualificare, marcare. Ovvero, se per esempio annotiamo una classe, permetteremo ad un software (per esempio il compilatore Java) di rendersi conto che tale classe è stata marcata, cosicché potrà implementare un certo comportamento di conseguenza. L'argomento è piuttosto complesso e più avanti studieremo il capitolo 13 che è interamente dedicato alle annotazioni. Comunque l'esempio più intuitivo di annotazione definita dal linguaggio stesso, riguarda proprio il problema di

implementazione dell'override che stavamo considerando. Esiste nel package `java.lang` l'annotazione **`Override`.** Questa la si può (e la si deve) utilizzare per marcare i metodi che vogliono essere un override di metodi ereditati. Per esempio:

```
public class PuntoTridimensionale extends Punto {
    @Override
    public double distanzadallOrigine() {
        //...
    }
}
```

Notiamo come l'utilizzo di un'annotazione richiede l'uso di un carattere che non avevamo mai incontrato nella sintassi Java: `@`.

Se marchiamo con `Override` il nostro metodo, nel caso violassimo una qualche regola dell'override, come nel seguente codice:

```
public class PuntoTridimensionale extends Punto {
    @Override
    public double distanzadallOrigine() {
        //...
    }
}
```

allora otterremmo un errore direttamente in compilazione:

```
method does not override a method from its superclass
    @Override public double distanzadallOrigine() {
     ^
1 error
```

Ottenere un errore in compilazione è molto meglio che averlo in fase di esecuzione.

8.3 Polimorfismo per dati

Il **polimorfismo per dati** permette essenzialmente di poter assegnare un reference di una superclasse ad un'istanza di una sottoclasse. Per esempio, tenendo conto che `PuntoTridimensionale` è sottoclasse di `Punto`, sarà assolutamente legale scrivere:

```
Punto ogg = new PuntoTridimensionale();
```

Il reference `ogg` infatti, punterà ad un indirizzo che valida il suo intervallo di puntamento. Praticamente l'interfaccia pubblica dell'oggetto creato (costituita dai metodi `setX()`, `getX()`, `setY()`, `getY()`, `setZ()`, `getZ()` e `distanzaDallOrigine()`) contiene l'interfaccia pubblica della classe `Punto` (costituita dai metodi `setX()`, `getX()`, `setY()`, `getX()` e `distanzaDallOrigine()`) e così il reference `ogg` "penserà" di puntare ad un oggetto `Punto`. Se volessimo rappresentare graficamente questa situazione, potremmo farlo come in figura 8.3.

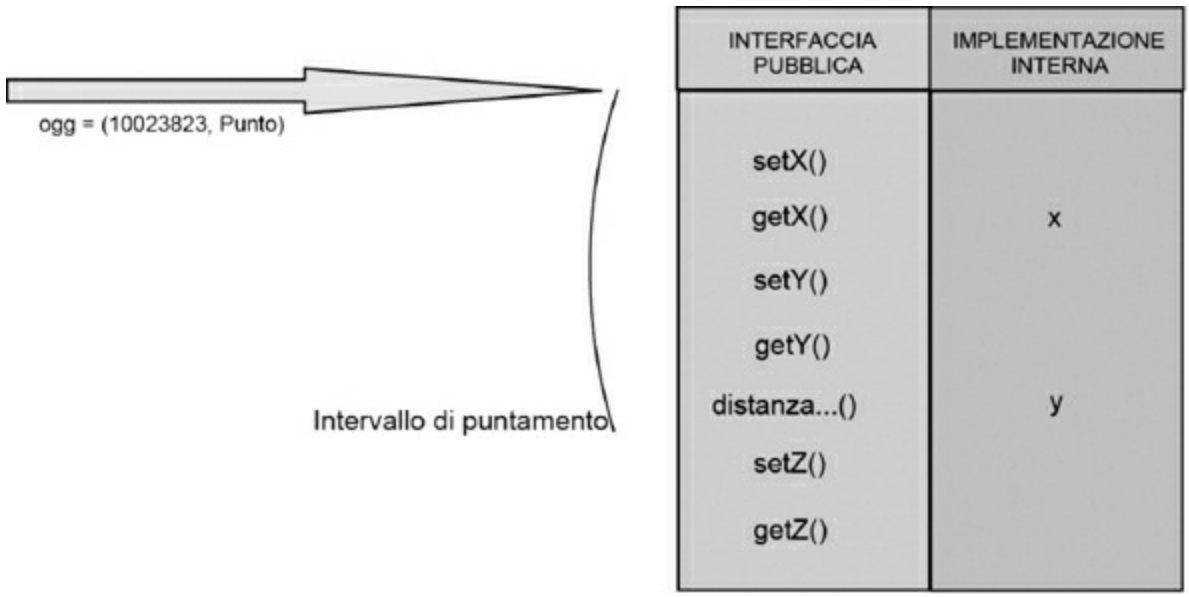


Figura 8.3 - Polimorfismo per dati.

Questo tipo di approccio ai dati ha però una limitazione. Un reference di una superclasse non potrà accedere ai campi dichiarati per la prima volta nella sottoclasse. Nell'esempio otterremmo un errore in compilazione se tentassimo di accedere ai metodi accessor e mutator della terza coordinata `z` del `PuntoTridimensionale` tramite il reference `ogg`. Ovvero, la codifica della seguente riga:

```
ogg.setZ(5);
```

produrrebbe un errore in fase di compilazione, dal momento che `ogg` è un reference che ha un intervallo di puntamento di tipo `Punto`.

A questo punto solitamente il lettore tende a chiedersi: “OK, ho capito che significa polimorfismo per dati, ma a che serve?”.

Terminata la lettura di questo capitolo avremo la risposta.

8.3.1 Parametri polimorfi

Sappiamo che i parametri in Java sono sempre passati per valore. Ciò implica che passare un parametro di tipo reference ad un metodo, significa passare il valore numerico del reference, in altre parole il suo indirizzo. A quest’indirizzo potrebbe risiedere un oggetto istanziato da una sottoclasse, grazie al polimorfismo per dati. In un metodo, un parametro di tipo reference si dice **parametro polimorfo** quando, anche essendo di fatto un reference relativo ad una determinata classe, può puntare ad un oggetto istanziato da una sottoclasse. Quindi, sfruttando il polimorfismo per dati, un parametro di un metodo potrebbe puntare ad oggetti diversi. È il caso del metodo `println()` che prende un parametro di tipo `Object`.

```
Punto p1 = new Punto();
System.out.println(p1);
```

Il lettore ha già appreso precedentemente che tutte le classi (compresa la classe `Punto`) sono sottoclassi di `Object`. Quindi potremmo invocare il metodo `println()` passandogli come parametro, anziché un’istanza di `Object`, un’istanza di `Punto` come `p1`. Ma al metodo `println()` che prende in input un `Object`, possiamo passare un’istanza di qualsiasi classe, dal momento che vale il polimorfismo per dati ed ogni classe è sottoclasse di `Object`.

Per adempiere allo scopo di “stampare un oggetto” il metodo `println()` chiamerà sul parametro passato in input il metodo `toString()`. Questo metodo di sicuro esiste, essendo stato dichiarato nella classe `Object` come vedremo nel paragrafo 8.3.7. Molte classi della libreria standard di Java eseguono un override di questo metodo, restituendo stringhe descrittive dell’oggetto. Se al metodo `println()` passassimo come parametro un oggetto di una classe che non ridefinisce il metodo `toString()`, sarebbe chiamato il metodo `toString()` ereditato dalla classe `Object` (come nel caso della classe `Punto`). Un’implementazione del metodo `toString()` per la classe `Punto` potrebbe essere la seguente:

```
@Override
public String toString() {
    return "(" + getX() + "," + getY() + ")";
}
```

8.3.2 Collezioni eterogenee

Una **collezione eterogenea** è una collezione composta da oggetti diversi (ad esempio un array di `Object` che in realtà immagazzina altri tipi di oggetti). Anche la possibilità di sfruttare collezioni eterogenee è garantita dal polimorfismo per dati. Infatti un array dichiarato di `Object` potrebbe contenere ogni tipo di oggetto:

```
Object arr[] = new Object[3];
arr[0] = new Punto(); //arr[0], arr[1], arr[2] sono
arr[1] = "Hello World!"; //reference ad Object che puntano
arr[2] = new Date(); //ad oggetti istanziati da sottoclassi
```

Il che equivale a scrivere:

```
Object arr[] = {new Punto(), "Hello World!", new Date()};
```

Presentiamo di seguito un esempio allo scopo di intuire l'utilità di questi concetti. Per semplicità ometteremo i metodi setter e getter nelle nostre classi. Immaginiamo di voler realizzare un sistema che stabilisca le paghe dei dipendenti di un'azienda, considerando le seguenti classi:

```
public class Dipendente {
    private String nome;
    private int stipendio;
    private int matricola;
    private String dataDiNascita;
    private String dataDiAssunzione;
    //metodi setter e getter omessi
}

public class Programmatore extends Dipendente {
    private String linguaggiConosciuti;
    private int anniDiEsperienza;
    //metodi setter e getter omessi
}

public class Dirigente extends Dipendente {
```

```
    private String orarioDiLavoro;
    //metodi setter e getter omessi
}

public class AgenteDiVendita extends Dipendente {
    private String [] portafoglioClienti;
    private int provvigioni;
    //metodi setter e getter omessi
}
...
```

Il nostro scopo è realizzare un metodo che stabilisca le paghe dei dipendenti. Potremmo ora utilizzare una collezione eterogenea di dipendenti ed un parametro polimorfo per risolvere il problema in modo semplice, veloce e sufficientemente elegante. Infatti, potremmo dichiarare una collezione eterogenea di dipendenti nel seguente modo:

```
Dipendente [] arr = new Dipendente [180];
arr[0] = new Dirigente();
arr[1] = new Programmatore();
arr[2] = new AgenteDiVendita();
...
```

Tra gli operatori di Java esiste un operatore binario che può essere utile in questi contesti: **`instanceof`**. Mediante questo costrutto si può testare a che tipo di oggetto punta un reference:

```
public void pagaDipendente(Dipendente dip) {
    if (dip instanceof Programmatore) {
        dip.setStipendio(1500);
    }
    else if (dip instanceof Dirigente) {
        dip.setStipendio(3000);
    }
    else if (dip instanceof AgenteDiVendita) {
        dip.stipendio = 1000 + getProvvigioni();
    }
    //...
}
```

Ora possiamo chiamare questo metodo all’interno di un ciclo foreach (di 180 iterazioni), passandogli tutti gli elementi della collezione eterogenea, e raggiungere così il nostro scopo:

```
...
for (Dipendente dipendente :  arr) {
    pagaDipendente(dipendente);
    //...
}
```

In particolare l'operatore `instanceof` ha come operandi un reference (il primo operando) e una classe (il secondo operando). Questo operatore restituisce `true` se il primo operando è un reference che punta ad un oggetto istanziato dal secondo operando o ad un oggetto istanziato da una sottoclasse specificata dal secondo operando. Se non si verifica una di queste due condizioni ritorna `false`. Ciò implica che se il metodo `pagaDipendente()` fosse scritto nel seguente modo:

```
public void pagaDipendente(Dipendente dip) {
    if (dip instanceof Dipendente) {
        dip.stipendio = 1000;
    }
    else if (dip instanceof Programmatore) {
...
```

tutti i dipendenti sarebbero pagati allo stesso modo. Infatti, anche se `dip` fosse un reference di tipo `Programmatore`, sarebbe comunque verificato il controllo del primo `if`, visto che se `dip` è un `Programmatore`, per l'ereditarietà è anche un `Dipendente`.

> **Come già accennato in precedenza, nella libreria standard esiste un nutrito gruppo di classi fondamentali per lo sviluppo, note sotto il nome di "collections". Si tratta di collezioni eterogenee e ridimensionabili come la classe `ArrayList` vista nella nota che conclude il paragrafo 8.2.2 sui varargs. Le collections rappresentano l'argomento principale del capitolo 14 relativo al package `java.util`.**

8.3.3 Casting di oggetti

Nell'esempio precedente abbiamo osservato che l'operatore `instanceof` ci permette di testare a quale tipo di istanza punta un reference. Ma già

sappiamo che il polimorfismo per dati, fa sì che il reference che punta ad un oggetto istanziato da una sottoclasse non possa accedere ai membri dichiarati nelle sottoclassi stesse. Esiste però la possibilità di ristabilire la piena accessibilità all'oggetto tramite il meccanismo del **casting di oggetti.**

Rifacciamoci all'esempio appena presentato. Supponiamo che lo stipendio di un programmatore dipenda dal numero di anni di esperienza. In questa situazione, dopo aver testato che il reference `dip` punta ad un'istanza di `Programmatore`, avremo bisogno di accedere alla variabile `anniDiEsperienza`. Se tentassimo di accedervi mediante la sintassi `dip.getAnniDiEsperienza()` otterremmo sicuramente un errore in compilazione. Ma se utilizzassimo il meccanismo del casting di oggetti supereremmo anche quest'ultimo ostacolo. Questo è possibile implementarlo dichiarando un reference a `Programmatore` e facendolo puntare all'indirizzo di memoria dove punta il reference `dip`, utilizzando il casting per confermare l'intervallo di puntamento. Il nuovo reference, essendo giusto, ci permetterà di accedere a qualsiasi membro dell'istanza di `Programmatore`.

Il casting di oggetti sfrutta una sintassi del tutto simile al casting tra dati primitivi:

```
if (dip instanceof Programmatore) {
    Programmatore pro = (Programmatore) dip;
    //...
```

Siamo ora in grado di accedere alla variabile `anniDiEsperienza` tramite la sintassi:

```
//...
if (pro.getAnniDiEsperienza() > 2)
//...
```

Come al solito, cerchiamo di chiarirci le idee cercando di schematizzare la situazione con una rappresentazione grafica. Diamo uno sguardo alla figura 8.4.

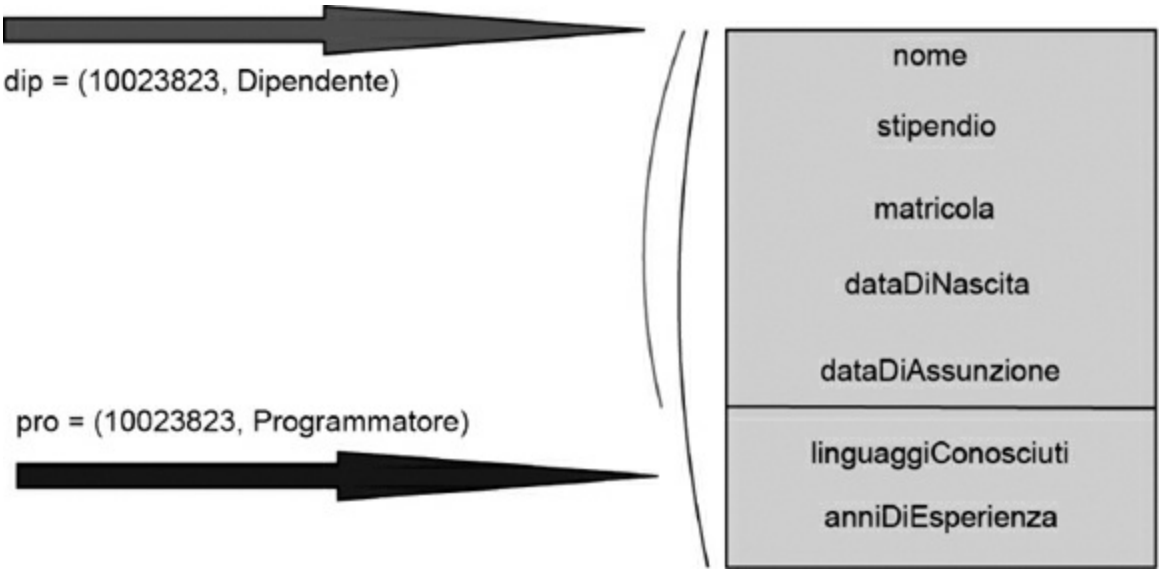


Figura 8.4 - Due diversi tipi di accesso per lo stesso oggetto.

Nella figura 8.4 possiamo notare come i due reference abbiano lo stesso valore numerico (indirizzo) ma differente intervallo di puntamento. Da questo dipende l'accessibilità all'oggetto.

Nel caso tentassimo di assegnare al reference `pro` l'indirizzo di `dip` senza l'utilizzo del casting, otterremmo un errore in compilazione ed un relativo messaggio che ci richiede un casting esplicito. Ancora una volta il comportamento del compilatore conferma la robustezza del linguaggio. Il compilatore non può stabilire se ad un certo indirizzo risiede un determinato oggetto piuttosto che un altro. È solo in esecuzione che la Java Virtual Machine può sfruttare l'operatore `instanceof` per risolvere il dubbio.

Il casting di oggetti non si deve considerare come strumento di programmazione standard, piuttosto come un utile mezzo per risolvere problemi progettuali. Una progettazione ideale farebbe a meno del casting di oggetti. Per quanto ci riguarda, all'interno di un progetto, la necessità del casting ci porta a pensare ad una forzatura e quindi ad un eventuale aggiornamento della progettazione. Tuttavia ci sono casi in cui è inevitabile l'utilizzo di questo strumento.

Ancora una volta osserviamo un altro aspetto che fa definire Java come linguaggio semplice da imparare. Il casting è un argomento che esiste anche in altri linguaggi e

riguarda i tipi di dati primitivi numerici. Esso viene realizzato troncando i bit in eccedenza di un tipo di dato il cui valore vuole essere forzato ad entrare in un altro tipo di dato più piccolo. Notiamo che, nel caso di casting di oggetti, non viene assolutamente troncato nessun bit, quindi si tratta di un processo completamente diverso! Se però ci astraiamo dai tipi di dati in questione, la differenza non sembra sussistere e Java permette di utilizzare la stessa sintassi, facilitando l'apprendimento e l'utilizzo al programmatore.

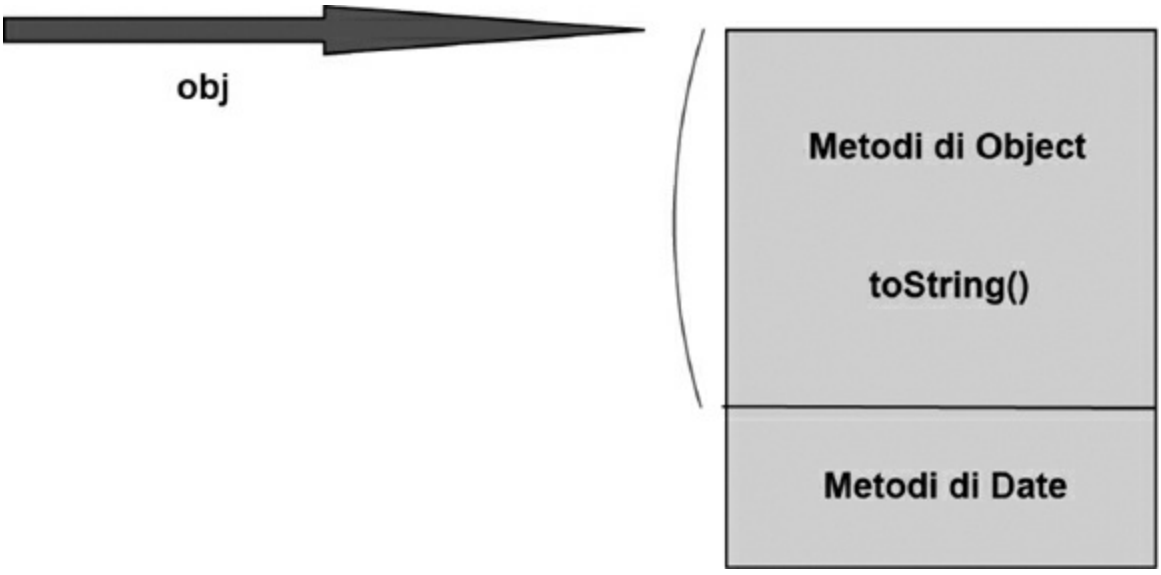


Figura 8.5 - Lo spazio dei metodi di un oggetto `Date` che sovrascrive `toString()` della classe Object.

8.3.4 Invocazione virtuale dei metodi

Un'invocazione ad un metodo `m` può definirsi **virtuale** quando `m` è definito in una classe `A`, ridefinito in una sottoclasse `B` (override) e invocato su un'istanza di `B`, tramite un reference di `A` (polimorfismo per dati). Quando s'invoca in maniera virtuale il metodo `m`, il compilatore "pensa" di invocare il metodo `m` della classe `A` (virtualmente). In realtà viene invocato il metodo ridefinito nella classe `B`. Un esempio classico è quello del metodo `toString()` della classe `Object`. Abbiamo già accennato al fatto che esso viene sottoposto a override in molte classi della libreria standard. Consideriamo la classe `Date`

del package `java.util`. In essa il metodo `toString()` è riscritto in modo tale da restituire informazioni sull'oggetto `Date` (giorno, mese, anno, ora, minuti, secondi, giorno della settimana, ora legale...). Consideriamo il seguente frammento di codice:

```
...
Object obj = new Date();
String s1 = obj.toString();
...
```

Il reference `s1` conterrà la stringa che contiene informazioni riguardo l'oggetto `Date`, unico oggetto istanziato. Possiamo vederne la schematizzazione in figura 8.5. Il reference `obj` può accedere solamente all'interfaccia pubblica della classe `Object` e quindi anche al metodo `toString()`. Il reference punta però a un'area di memoria dove risiede un oggetto della classe `Date`, nella quale il metodo `toString()` ha una diversa implementazione rispetto a quella che fornisce la classe `Object`.

Il metodo `toString()` era già stato chiamato in maniera virtuale nell'esempio del paragrafo sui parametri polimorfi.

8.3.5 Esempio d'utilizzo del polimorfismo

Supponiamo di avere a disposizione le seguenti classi:

```
public abstract class Veicolo {
    public abstract void accelera();
    public abstract void decelera();
}

public class Aereo extends Veicolo {
    public void decolla() {
        //...
    }
    public void atterra() {
        //...
    }
```

```
    @Override
    public void accelera() {
        //...
    }
    @Override
    public void decelera() {
        //...
    }
    //...
}

public class Automobile extends Veicolo {
    @Override
    public void accelera() {
        //...
    }
    @Override
    public void decelera() {
        //...

    }
    public void innestaRetromarcia() {
        //...
    }
    //...
}

public class Nave extends Veicolo {
    @Override
    public void accelera() {
        //...
    }
    @Override
    public void decelera() {
        //...
    }
    public void gettaAncora() {
        //...
    }
    //...
}
```

La superclasse `Veicolo` definisce i metodi `accelera` e `decelera`, che vengono poi ridefiniti in sottoclassi più specifiche quali `Aereo`, `Automobile`

e `Nave`.

> **Si noti che i metodi sono stati dichiarati astratti. Effettivamente, come accelera un veicolo? Come una nave o come un aereo? La risposta più giusta è che non c'è risposta! È indefinita. Un veicolo sicuramente può accelerare e decelerare ma, in questo contesto, non si può dire come. La soluzione più giusta appare quindi dichiarare i metodi astratti.**

Consideriamo la seguente classe, che fa uso dell'overload:

```
public class Viaggiatore {
    public void viaggia(Automobile a) {
        a.accelera();
        //...
    }
    public void viaggia(Aereo a) {
        a.accelera();
        //...
    }
    public void viaggia(Nave n) {
        n.accelera();
        //...
    }

    //...
}
```

Nonostante l'overload rappresenti una soluzione notevole per la codifica della classe `Viaggiatore`, notiamo una certa ripetizione nei blocchi di codice dei tre metodi. Sfruttando infatti un parametro polimorfo ed un metodo virtuale, la classe `Viaggiatore` potrebbe essere codificata in modo più compatto e funzionale:

```
public class Viaggiatore {
    public void viaggia(Veicolo v) { //parametro Polimorfo
        v.accelera(); //invocazione metodo virtuale
        //...
```

```
    }
    //...
}
```

Il seguente frammento di codice infatti, utilizza le precedenti classi:

```
Viaggiatore claudio = new Viaggiatore();
Automobile fiat500 = new Automobile();
// avremmo potuto istanziare anche una Nave o un Aereo
claudio.viaggia(fiat500);
```

Notiamo la chiarezza e la versatilità del codice. Ci siamo calati in un contesto altamente estensibile: se volessimo introdurre un nuovo `Veicolo` (supponiamo la classe `Bicicletta`) basterebbe codificarla senza toccare quanto scritto finora! Anche il seguente codice costituisce un valido esempio:

```
Viaggiatore claudio = new Viaggiatore();
Aereo piper = new Aereo();
claudio.viaggia(piper);
```

8.3.6 Polimorfismo e interfacce

Il polimorfismo per dati funziona anche con le interfacce. Questo significa che possiamo usare reference di interfacce per puntare ad oggetti che implementano queste interfacce. Nel capitolo precedente abbiamo asserito che le interfacce possono essere definite per astrarre comportamenti e aggettivi, per poi essere implementati in classi concrete. Nel capitolo precedente avevamo definito l'interfaccia `Volante` in questo modo:

```
public interface Volante {
    void atterra();
    void decolla();
}
```

Ogni classe che deve astrarre un concetto di oggetto volante (come un aereo, un elicottero o un uccello) deve implementare l'interfaccia. Quindi avevamo riscritto la classe `Aereo` nel seguente modo:

```
public class Aereo extends Veicolo implements Volante {
    @Override
    public void atterra() {
        // override del metodo di Volante
```

```
    }
    @Override
    public void decolla() {
        // override del metodo di Volante
    }
    @Override
    public void accelera() {
        // override del metodo di Veicolo
    }
    @Override
    public void decelera() {
        // override del metodo di Veicolo
    }
}
```

Potremmo quindi creare parametri polimorfi per sfruttare l'interfaccia `Volante`:

```
public class TorreDiControllo {
    public void autorizzaAtterraggio(Volante v) {
        v.atterra();
    }

    public void autorizzaDecollo(Volante v) {
        v.decolla();
    }
}
```

Così facendo possiamo passare a questi metodi oggetti volanti creati da classi che implementano l'interfaccia `Volante`.

Riepilogo

In questo capitolo abbiamo soprattutto esplorato il supporto di Java al polimorfismo dividendolo in vari sottoargomenti. Il polimorfismo si divide in polimorfismo per metodi e per dati. Il **polimorfismo per metodi** è un concetto che trova una sua implementazione in Java tramite overload ed override.

L'**overload** permette di creare metodi con lo stesso nome (ma differente lista dei parametri) nella stessa classe. La coppia nome, lista di parametri, è detta **firma** del metodo. Un'alternativa all'overload può essere a volte

costituita dall'uso di metodi che sfruttano varargs, ma bisogna valutare di volta in volta costi e benefici.

L'**override** consente di riscrivere nelle sottoclassi i metodi ereditati dalle superclassi. Rispetto al metodo originale della superclasse, il metodo riscritto deve avere la stessa firma, lo stesso tipo di ritorno e non deve essere meno accessibile. L'annotazione `Override` ci permette di marcare i metodi riscritti in modo tale da dichiarare al compilatore che stiamo riscrivendo un metodo ereditato. In tal caso, il compilatore controllerà per noi se stiamo applicando correttamente l'override, restituendoci un eventuale errore se ce ne fosse bisogno.

Il **polimorfismo per dati** ha di per sé una implementazione in Java. Con esso possiamo far puntare un reference di una superclasse ad un oggetto istanziato da una sottoclasse. La vera utilità del polimorfismo per dati si evince dall'utilizzo di parametri polimorfi, invocazioni virtuali dei metodi, e dalle collezioni eterogenee.

Una **collezione eterogenea** è una collezione di dati, per esempio un array che contiene oggetti diversi. Per quanto riguarda gli array, questo significa che l'array viene dichiarato di un certo tipo, ma poi viene popolato da oggetti istanziati da sottoclassi di tale tipo. Esistono delle classi nella libreria standard di Java (cfr. capitolo 14) che vengono chiamate *collection*, che sono potenzialmente eterogenee. Le *collection* in realtà sono molto più utilizzate rispetto agli array, perché ridimensionabili.

I **parametri polimorfi** sono parametri di metodi dichiarati di un certo tipo, che poi al runtime possono puntare a oggetti istanziati da sottoclassi di tale tipo.

Inoltre l'utilizzo contemporaneo del polimorfismo per dati e dell'override dà luogo alla possibilità di **invocare metodi in maniera virtuale.** Con un'unica istruzione che invoca un metodo su una certa classe, potremo quindi invocare dinamicamente metodi riscritti su oggetti istanziati da sottoclassi.

Tutti gli strumenti di programmazione che abbiamo visto in questo capitolo sono molto utili ed un loro corretto utilizzo deve diventare per il lettore uno degli obiettivi fondamentali. Chiunque riesce a smanettare con Java, ma ci sono altri linguaggi più adatti per smanettare. Lo sviluppo con Java va accompagnato dalla ricerca di soluzioni di progettazione per agevolare la programmazione. Se non si desidera fare questo, forse Java non

è il linguaggio giusto da imparare. Non sarà sicuramente semplice utilizzare correttamente i potenti strumenti della programmazione ad oggetti. Nonostante al lettore potrebbero essere chiari tutti gli argomenti che ha studiato sinora, non è detto che sia in grado di creare un programma iniziando da zero, anzi è altamente improbabile che ci riesca se non ha precedenti esperienze di programmazione. Certamente può aiutare molto conoscere una metodologia object oriented (consultare l'appendice on line "Bibliografia") o almeno UML (a cui è dedicata una delle appendici on line). L'esperienza rimane come sempre la migliore palestra. Il consiglio per chi inizia è quindi quello di provare a creare un proprio progetto personale che potrebbe essergli utile, ovviamente non troppo complicato, ed eseguire gli esercizi forniti dal libro. In questo modo toccherà con mano le difficoltà della progettazione, ed anche se il progetto rischia di non vedere la luce, almeno si inizierà a fare esperienza, si faranno degli errori e si imparerà a correggerli.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere il significato del polimorfismo (unità 8.1)	☐	
Saper utilizzare l'overload, l'override ed il polimorfismo per metodi (unità 8.2 e 8.3)	☐	
Comprendere e saper utilizzare il polimorfismo per dati con le sue implementazioni: le collezioni eterogenee, i parametri polimorfi e le invocazioni virtuali di metodi (unità 8.3)	☐	
Sapere utilizzare l'operatore instanceof ed il casting di oggetti (unità 8.3)	☐	

Note:

9

Eccezioni ed asserzioni

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Comprendere le differenze tra eccezioni, errori ed asserzioni (unità 9.1).
- ✔ Saper gestire le varie tipologie di eccezioni con i blocchi *try-catch* (unità 9.2, 9.3).
- ✔ Saper utilizzare il costrutto *try with resources* (unità 9.4).
- ✔ Capire e saper utilizzare le asserzioni (unità 9.6).

I concetti relativi ad eccezioni, errori ed asserzioni e le relative gestioni permettono allo sviluppatore di scrivere software robusto, ovvero che funziona correttamente anche in situazioni impreviste. Nelle prossime pagine parleremo non solo di una parte della sintassi fondamentale di Java, ma anche di progettazione. In particolare il meccanismo di propagazione delle eccezioni è una caratteristica del linguaggio da cui non si può prescindere. Le asserzioni invece sono usate raramente, ma sono un ottimo strumento per scrivere codice funzionante.

9.1 Eccezioni, errori ed asserzioni

Dei tre argomenti di cui tratta questo capitolo, il più importante è sicuramente la gestione delle eccezioni, vero e proprio punto cardine del linguaggio. È possibile definire un'**eccezione** come una situazione imprevista che può presentarsi durante il flusso di un'applicazione. È possibile gestire un'eccezione in Java imparando ad utilizzare cinque semplici parole chiave: `try`, `catch`, `finally`, `throw` e `throws`. Sarà anche possibile creare eccezioni personalizzate e decidere non solo come, ma anche in quale parte del codice gestirle, grazie ad un meccanismo di propagazione estremamente potente. Questo concetto è implementato nella libreria Java mediante la classe `Exception` e le sue sottoclassi. Un esempio di eccezione che potrebbe verificarsi all'interno di un programma è quello relativo ad una divisione tra due variabili numeriche nella quale la variabile divisore ha valore `0`. Come è noto, infatti, tale operazione non è eseguibile. È invece possibile definire un **errore** come una situazione imprevista non dipendente da un errore commesso dallo sviluppatore. A differenza delle eccezioni, quindi, gli errori non sono gestibili. Questo concetto è implementato nella libreria Java mediante la classe `Error` e le sue sottoclassi. Un esempio di errore che potrebbe causare un programma è quello relativo alla terminazione delle risorse di memoria. Questa condizione non è gestibile. Inoltre, è possibile definire un'**asserzione** come una condizione che deve essere verificata affinché lo sviluppatore consideri corretta una parte di codice. A differenza delle eccezioni e degli errori, le asserzioni rappresentano uno strumento per testare la robustezza del software. Si possono abilitare in fase di sviluppo e test, ed eventualmente disabilitare in fase di rilascio. Così l'esecuzione del software non subirà alcun tipo di rallentamento. Questo concetto è implementato tramite la parola chiave `assert`. Per esempio un'asserzione potrebbe asserire che la variabile con funzione di divisore in una divisione debba essere diversa da `0`. Se questa condizione non dovesse verificarsi l'esecuzione del codice verrà interrotta.

9.2 Gerarchie e categorizzazioni

Nella libreria standard di Java esiste una gerarchia di classi che mette in relazione la classe **`Exception`** e la classe **`Error`**. Entrambe queste classi estendono la superclasse **`Throwable`**, come si può vedere in Figura 9.1.

Un'ulteriore categorizzazione delle eccezioni è data dalla divisione delle eccezioni in checked ed unchecked exception. Ci si riferisce alle `RuntimeException` (e alle sue sottoclassi) come unchecked exception. Tutte le altre eccezioni (ovvero tutte quelle che non estendono `RuntimeException`), sono dette checked exception. Se si utilizza un metodo che lancia una checked exception senza gestirla da qualche parte, la compilazione non andrà a buon fine. Da qui il termine checked exception (in italiano "eccezioni controllate").

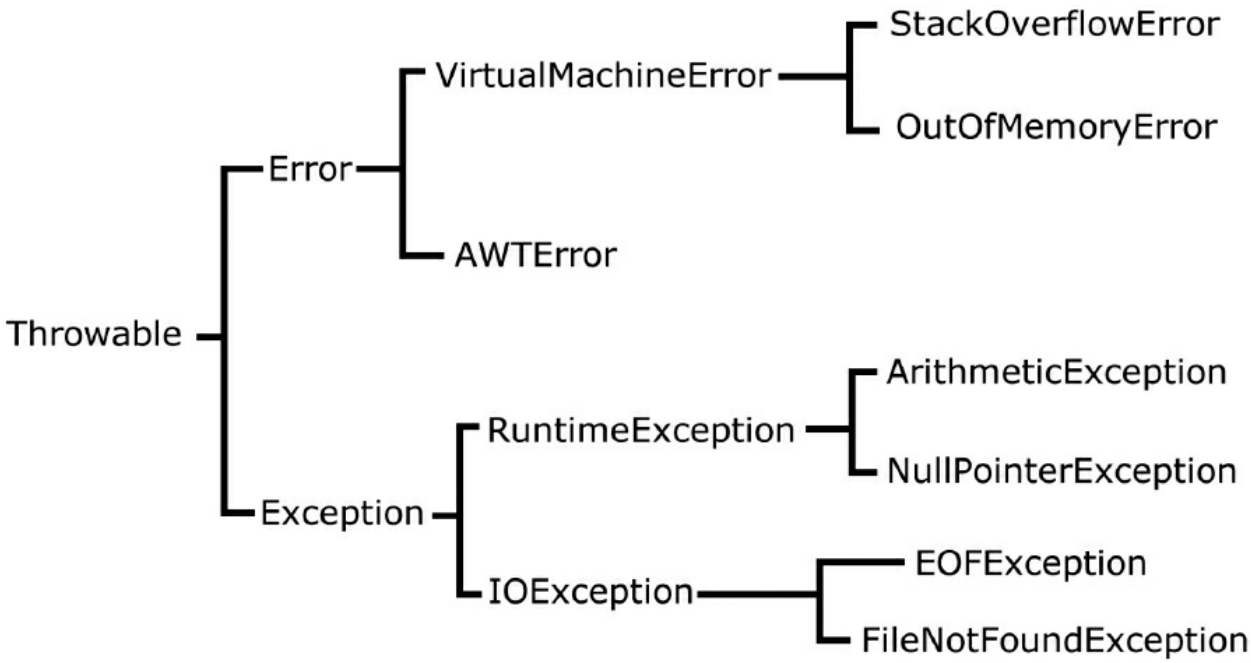


Figura 9.1 - Gerarchia per le classi Throwable.

Come già detto, non bisognerà fare confusione tra il concetto di errore (problema che un programma non può risolvere) e di eccezione (problema non critico gestibile). Il fatto che sia la classe `Exception` sia la classe `Error` estendano una classe che si chiama "lanciabile" (`Throwable`) è dovuto al meccanismo con cui la Java Virtual Machine reagisce quando si imbatte in una eccezione-errore. Infatti, se il nostro programma genera un'eccezione durante il runtime, la JVM istanzia un oggetto della classe eccezione relativa al problema e *lancia* l'eccezione appena istanziata (tramite la parola chiave

`throw`). Se il nostro codice non *cattura* (tramite la parola chiave `catch`) l'eccezione, il gestore automatico della JVM interromperà il programma generando in output informazioni dettagliate su ciò che è accaduto. Per esempio supponiamo che durante l'esecuzione un programma provi ad eseguire una divisione per zero tra interi. La JVM istanzierà un oggetto di tipo `ArithmeticException` (inizializzandolo opportunamente) e lo lancerà. In pratica è come se la JVM eseguisse le seguenti righe di codice:

```
ArithmeticException exc = new ArithmeticException();
throw exc;
```

Tutto avviene dietro le quinte e sarà trasparente allo sviluppatore.

9.3 Meccanismo per la gestione delle eccezioni

Come già asserito in precedenza, lo sviluppatore ha a disposizione alcune parole chiave per gestire le eccezioni: **`try`**, **`catch`**, **`finally`**, **`throw`** e **`throws`**. Se bisogna sviluppare una parte di codice che potenzialmente può scatenare un'eccezione, è possibile circondarla con un blocco `try` seguito da uno o più blocchi `catch`. Per esempio:

```
public class Ecc1 {
    public static void main(String args[]) {
        int a = 10;
        int b = 0;
        int c = a/b;
        System.out.println(c);
    }
}
```

Questa classe può essere compilata senza problemi, ma genererà un'eccezione durante la sua esecuzione, dovuta all'impossibilità di eseguire una divisione per zero. In tal caso la JVM, dopo aver interrotto il programma, produrrà il seguente output:

```
Exception in thread "main" java.lang.ArithmeticException: / by
zero
  at Ecc1.main(Ecc1.java:6)
```

Un messaggio di sicuro molto esplicativo dal momento che sono stati evidenziati:

- il tipo di eccezione (`java.lang.ArithmeticException`);
- un messaggio descrittivo (`/ by zero`);
- il metodo in cui è stata lanciata l'eccezione (`at Ecc1.main`);
- il file in cui è stata lanciata l'eccezione (Ecc1.java);
- la riga in cui è stata lanciata l'eccezione (`:6`).

L'unico problema è che il programma è terminato prematuramente. Utilizzando le parole chiave `try` e `catch` sarà possibile gestire l'eccezione in maniera personalizzata, facendo terminare il programma senza che sia interrotto bruscamente:

```
public class Ecc2 {
    public static void main(String args[]) {
        int a = 10;
        int b = 0;
        try {
            int c = a/b;
            System.out.println(c);
        }
        catch (ArithmeticException exc) {
            System.out.println("Divisione per zero...");
        }
    }
}
```

Quando la JVM eseguirà tale codice incontrerà la divisione per zero della prima riga del blocco `try` e lancerà l'eccezione `ArithmeticException`, che verrà catturata nel blocco `catch` seguente. Quindi non sarà eseguita la riga che doveva stampare la variabile `c`, bensì quella che stampa la stringa "`Divisione per zero...`" con la quale abbiamo gestito l'eccezione ed abbiamo permesso al nostro programma di terminare in maniera naturale. Come il lettore avrà sicuramente notato la sintassi dei blocchi `try - catch` è piuttosto strana, ma presto ci si fa l'abitudine, perché è presente più volte praticamente in tutti i programmi Java. In particolare il blocco `catch` deve

dichiarare un parametro (come se fosse un metodo) del tipo dell'eccezione che deve essere catturata. Nell'esempio precedente il reference `exc` puntava proprio all'eccezione che la JVM aveva istanziato e lanciato. Infatti, tramite esso è possibile reperire informazioni proprio sull'eccezione stessa. Il modo più semplice per ottenere informazioni su ciò che è successo è invocare il metodo `printStackTrace()` sull'eccezione:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    exc.printStackTrace();
}
```

Il metodo `printStackTrace()` produrrà in output i messaggi informativi di cui sopra, che il programma avrebbe prodotto se l'eccezione non fosse stata gestita, ma senza interrompere il programma stesso. È fondamentale che si dichiari tramite il blocco `catch`, un'eccezione del tipo giusto. Per esempio, il seguente frammento di codice:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (NullPointerException exc) {
    exc.printStackTrace();
}
```

produrrebbe un'eccezione non gestita e quindi un'immediata terminazione del programma. Infatti il blocco `try` non ha mai lanciato una `NullPointerException`, ma una `ArithmeticException`. Come per i metodi, anche per i blocchi `catch` i parametri possono essere **polimorfi.** Per esempio, il seguente frammento di codice:

```
int a = 10;
```

```
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

contiene un blocco `catch` che gestirebbe qualsiasi tipo di eccezione, essendo `Exception` la superclasse da cui discende ogni altra eccezione. Il reference `exc` è in questo esempio un parametro polimorfo. È anche possibile far seguire ad un blocco `try` più blocchi `catch`, come nel seguente esempio:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

In questo modo il nostro programma risulterebbe più robusto e gestirebbe diversi tipi di eccezioni. Nel peggiore dei casi, ovvero se il blocco `try` lanciasse un'eccezione non prevista, l'ultimo blocco `catch` gestirebbe il problema.

È fondamentale l'ordine dei blocchi `catch`. Se avessimo:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (Exception exc) {
```

```
    exc.printStackTrace();
 }
 catch (ArithmeticException exc) {
    System.out.println("Divisione per zero...");
 }
 catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
 }
```

gli ultimi due `catch` sarebbero superflui e il compilatore segnalerebbe l'errore nel seguente modo:

```
D:\Ecc2.java:12: exception java.lang.ArithmeticException has
already
  been caught
        catch (ArithmeticException exc) {
        ^
D:\Ecc2.java:15: exception java.lang.NullPointerException has
already
  been caught
        catch (NullPointerException exc) {
        ^
2 errors
```

Ci sono casi in cui il contenuto dei blocchi `catch` è identico, nonostante servano per catturare eccezioni diverse, per esempio:

```
 int a = 10;
 int b = 0;
 try {
    int c = a/b;
    System.out.println(c);
 }
 catch (ArithmeticException exc) {
    System.out.println(exc.getMessage());
 }
 catch (NullPointerException exc) {
    System.out.println(exc.getMessage());
 }
 catch (Exception exc) {
    exc.printStackTrace();
 }
```

Per evitare ridondanze è possibile dichiarare in un unico `catch`, più di un

parametro come mostrato di seguito:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException | NullPointerException exc) {
    System.out.println(exc.getMessage());
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

Il simbolo "|" viene utilizzato per separare i nomi delle classi eccezione. Questo tipo di gestione di eccezioni cumulativa, prende il nome di **multi catch**, e fu introdotto per la prima volta con la versione 7 di Java.

È possibile far seguire ad un blocco `try`, oltre a blocchi `catch`, un altro blocco definito dalla parola chiave `finally`, per esempio:

```
public class Ecc4 {
    public static void main(String args[]) {
        int a = 10;
        int b = 0;
        try {
            int c = a/b;
            System.out.println(c);
        }
        catch (ArithmeticException exc) {
            System.out.println("Divisione per zero...");
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
        finally {
            System.out.println("Operazione terminata");
        }
    }
}
```

Ciò che è definito in un blocco `finally` viene eseguito in qualsiasi caso, sia se viene lanciata l'eccezione, sia se non viene lanciata. Per esempio, è

possibile utilizzare un blocco `finally` quando esistono operazioni critiche che devono essere eseguite in qualsiasi caso. L'output del precedente programma è:

```
Divisione per zero...
Operazione terminata
```

Se invece la variabile `b` fosse impostata a `2` piuttosto che a `0`, l'output sarebbe:

```
5
Operazione terminata
```

Un classico esempio (più significativo del precedente) in cui la parola `finally` è spesso utilizzata è il seguente:

```
 public void selectFromDB() {
     Connection conn = null;
     Statement stmt = null;
     ResultSet rs = null;
     try {
         conn = DriverManager.getConnection(url, username,
 password);

         stmt = conn.createStatement();
         rs = stmt.executeQuery("SELECT * FROM PERSONA");
         while (rs.next()) {
             System.out.println(rs.getString(1));
         } catch (SQLException e) {
             e.printStackTrace();
         } finally {
             if (rs != null) {
                 try {
                     rs.close();
                 } catch (SQLException e) {
                     e.printStackTrace();
                 }
             }
             rs = null;
             if (stmt != null) {
                 try {
                     stmt.close();
                 } catch (SQLException e) {
                     e.printStackTrace();
```

```
                }
            }
            stmt = null;
            if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
            conn = null;
        }
    }
}
```

Il metodo precedente tenta di eseguire una query verso un database, tramite le interfacce JDBC offerte dal package `java.sql`. Nell'esempio `stmt` è un oggetto `Statement` e `conn` è un oggetto di tipo `Connection`. Il comando `executeQuery()` esegue la query passata in input. Se ci sono problemi (per esempio sintassi SQL scorretta, chiave primaria già presente etc.) la JVM lancerà una `SQLException`, che sarà catturata nel relativo blocco `catch`. In ogni caso, dopo il tentativo di interrogazione la connessione al database deve essere chiusa, così come anche l'oggetto `Statement` e l'oggetto `ResultSet` (per maggiori approfondimenti su JDBC rimandiamo il lettore all'appendice R oppure per una breve introduzione all'indirizzo http://www.claudiodesio.com/java/jdbc.htm); questo è garantito dal codice implementato nel blocco `finally`.

> **È possibile anche far seguire direttamente ad un blocco `try` un blocco `finally`. Quest'ultimo sarà eseguito sicuramente dopo l'esecuzione del blocco `try`, sia se l'eccezione viene lanciata, sia se non viene lanciata. Comunque, se l'eccezione fosse lanciata, non essendo gestita con un blocco `catch`, il programma terminerebbe anormalmente.**

9.4 Try with resources

Dalla versione 7 di Java, alcune classi ed interfacce (tra cui `Connection`)

sono state riviste per supportare il meccanismo del cosiddetto **try with resources.** Questo consente la chiusura automatica degli oggetti che necessiterebbero di essere chiusi, una volta utilizzati. La sintassi prevede la dichiarazione dell'oggetto (o degli oggetti) da chiudere automaticamente come parametri del blocco `try`. Segue un esempio (equivalente al precedente):

```
try(Connection conn =
  DriverManager.getConnection(url, username, password);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM PERSONA
")) {

    while (rs.next()) {
        System.out.println(rs.getString(1));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

Gli oggetti chiudibili `conn`, `stmt` e `rs`, sono quindi dichiarati come se fossero parametri del blocco `try` (che per l'occasione dovrebbe essere chiamato blocco *try with resources).* Quando il blocco terminerà la sua esecuzione, le tre risorse verranno automaticamente chiuse. Questo avverrà sia nel caso positivo (il codice viene eseguito correttamente), sia nel caso negativo (per esempio viene lanciata un'eccezione). Esattamente come se i comandi per chiudere le risorse si trovassero all'interno di una clausola `finally`. Si tratta di un vantaggio non da poco.

> **Anche con un blocco try with resources è possibile dichiarare un numero arbitrario di blocchi** `catch` **ed un** `finally`**, come se si trattasse di un** `try` **ordinario. Questi blocchi verranno eseguiti solo dopo che le risorse saranno state chiuse.**

Se durante l'esecuzione del blocco `try` fosse lanciata un'eccezione (per esempio se la query avesse una sintassi sbagliata), questa verrà considerata

prioritaria rispetto ad eventuali eccezioni lanciate automaticamente dal blocco *try with resources* (nel caso si presentasse qualche problema nel chiudere le risorse utilizzate). In tal caso la JVM lancerà l'eccezione applicativa interna al blocco `try`, e dichiarerà *suppressed* le eccezioni lanciate durante la chiusura automatica degli oggetti utilizzati (ricordiamo che tali oggetti verranno chiusi come se si trovassero in una clausola `finally`). Queste eccezioni saranno comunque in qualche modo impostate come attributi dell'eccezione applicativa lanciata, e sarà possibile accedervi chiamando il metodo `getSuppressed()` sull'oggetto eccezione che viene lanciato.

Non tutti gli oggetti sono chiudibili. In particolare gli oggetti che devono essere chiusi sono oggetti che implementano l'interfaccia `AutoCloseable` (package `java.lang`) o l'interfaccia `Closeable` (package `java.io`). Per esempio la seguente classe è chiudibile:

```
public class RisorsaChiudibile implements AutoCloseable {
    public void close() {
        System.out.println("Chiamato metodo close()");
    }

    public String toString() {
        return "RisorsaChiudibile";
    }
}
```

Quindi può essere utilizzata in un costrutto *try with resources* nel seguente modo:

```
public class TestTryWithResource {
    public static void main(String args[]) {
        try (RisorsaChiudibile risorsaChiudibile = new
RisorsaChiudibile()) {
            System.out.println("Sto per chiudere: " +
risorsaChiudibile);
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
    }
}
```

Il cui output sarà:

```
Sto per chiudere: RisorsaChiudibile
Chiamato metodo close()
```

Con Java 9 è stata introdotta una novità rispetto al passato. Ora è possibile utilizzare all'interno delle parentesi tonde del *try with resources* anche variabili `final` oppure **effettivamente non modificate** (in inglese **effectively final).** Con "variabili effettivamente non modificate" intendiamo variabili che anche non essendo dichiarate `final`, non vengono modificate. Il compilatore è in grado di distinguere se una variabile è effettivamente non modificata o meno. Quindi il metodo `main()` dell'esempio precedente, potrà essere scritto in modo equivalente come di seguito:

```
    public static void main(String args[]) {
        RisorsaChiudibile risorsaChiudibile = new
RisorsaChiudibile();
        try (risorsaChiudibile) {
            System.out.println("Sto per chiudere: " +
risorsaChiudibile);
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
    }
```

Così il costrutto del *try with resources* diventa più conciso e leggibile.

9.5 Eccezioni personalizzate e propagazione dell'eccezione

Il lettore con meno esperienza di programmazione potrebbe chiedersi in questo momento perché utilizzare il meccanismo delle eccezioni piuttosto che delle semplici condizioni `if`. Nei prossimi paragrafi cercheremo di capire le differenze. Ci sono alcune tipologie di eccezioni che sono più frequenti e quindi più conosciute dagli sviluppatori Java. Si tratta di:

- `NullPointerException`: probabilmente la più frequente tra le eccezioni. Viene lanciata dalla JVM quando per esempio viene chiamato un metodo su di un reference che invece punta a `null`.

- `ArrayIndexOutOfBoundsException`: questa eccezione viene lanciata quando si prova ad accedere ad un array con un indice che è troppo alto rispetto alle sue dimensioni, o minore di zero.
- `ClassCastException`: eccezione particolarmente insidiosa; è lanciata al runtime quando si prova ad effettuare un cast verso un tipo di classe sbagliato.

Queste eccezioni appartengono tutte al package `java.lang`. Inoltre, se si utilizzano altri package come `java.io`, bisognerà gestire spesso le eccezioni come `IOException` e le sue sottoclassi (`FileNotFoundException`, `EOFException` etc.). Stesso discorso con la libreria `java.sql` e l'eccezione `SQLException`, il package `java.net` e la `ConnectException` e così via. Lo sviluppatore imparerà con l'esperienza come gestire tutte queste eccezioni. È però altrettanto probabile che qualche volta occorra definire nuovi tipi di eccezione. Infatti, per un particolare programma, potrebbe essere una eccezione anche una divisione per 5. Più verosimilmente, un programma che deve gestire in maniera automatica le prenotazioni per un teatro potrebbe voler lanciare un'eccezione nel momento in cui si tenti di prenotare un posto non più disponibile. In tal caso la soluzione è estendere la classe `Exception` ed eventualmente aggiungere membri e effettuare override di metodi come `toString()`. Segue un esempio:

```
public class PrenotazioneException extends Exception {
    public PrenotazioneException() {
        // Il costruttore di Exception chiamato inizializza
la
        // variabile privata message
        super("Problema con la prenotazione");
    }
    public String toString() {
        return getMessage() + ": posti esauriti!";
    }
}
```

La nostra eccezione contiene informazioni sul problema e rappresenta una astrazione corretta. Tuttavia la JVM non può lanciare automaticamente una `PrenotazioneException` nel caso si tenti di prenotare un posto non più disponibile. La JVM, infatti, sa quando lanciare una `ArithmeticException`

ma non sa quando lanciare una `PrenotazioneException`. In tal caso sarà compito dello sviluppatore lanciare l'eccezione. Esiste infatti la parola chiave `throw` (in inglese "lancia") che permette il lancio di un'eccezione tramite la seguente sintassi:

```
PrenotazioneException exc = new PrenotazioneException();
throw exc;
```

o equivalentemente (dato che il reference `exc` poi non sarebbe più utilizzabile):

```
throw new PrenotazioneException();
```

L'istruzione `throw` causa l'immediata uscita dal metodo in cui ci si trova. Il lancio dell'eccezione dovrebbe seguire un controllo condizionale come il seguente:

```
if (postiDisponibili == 0) {
    throw new PrenotazioneException();
}
```

Il codice precedente farebbe terminare prematuramente il programma a meno di gestire l'eccezione come segue:

```
try {
    // controllo sulla disponibilità dei posti

    if (postiDisponibili == 0) {
        //lancio dell'eccezione
        throw new PrenotazioneException();
    }
    // istruzione eseguita
    // se non viene lanciata l'eccezione
    postiDisponibili--;
}
catch (PrenotazioneException exc) {
    System.out.println(exc.toString());
}
```

Il lettore avrà sicuramente notato che il codice precedente non rappresenta un buon esempio di gestione dell'eccezione: dovendo utilizzare la condizione `if`, sembra infatti superfluo l'utilizzo dell'eccezione. In effetti è così! Ma ci

deve essere una ragione per la quale esiste la possibilità di creare eccezioni personalizzate e di poterle lanciare. Questa ragione è la **propagazione dell'eccezione** per i metodi chiamanti. La potenza della gestione delle eccezioni è dovuta essenzialmente a questo meccanismo di propagazione. Per comprenderlo bene affidiamoci ad un ulteriore esempio. Supponiamo di avere la seguente classe:

```
public class Botteghino {
  private int postiDisponibili;

    public Botteghino() {
        postiDisponibili = 100;
    }

    public void prenota() {
        try {
            // controllo sulla disponibilità dei posti
            if (postiDisponibili == 0) {
              //lancio dell'eccezione
              throw new PrenotazioneException();
            }
            // metodo che realizza la prenotazione
            // se non viene lanciata l'eccezione
            postiDisponibili--;
        }
        catch (PrenotazioneException exc) {
            System.out.println(exc.toString());
        }
    }
}
```

La classe `Botteghino` astrae in maniera semplicistica un botteghino virtuale che permette di prenotare i posti in un teatro. Ora consideriamo la seguente classe eseguibile (con metodo `main()`) che utilizza la classe `Botteghino`:

```
public class GestorePrenotazioni {
    public static void main(String [] args) {
        Botteghino botteghino = new Botteghino();
        for (int i = 1; i <= 101; ++i) {
            botteghino.prenota();
            System.out.println("Prenotato posto n° " + i);
        }
    }
```

```
}
```

Per una classe del genere, il fatto che l'eccezione sia gestita all'interno della classe `Botteghino` rappresenta un problema. Infatti l'output del programma sarà:

```
Prenotato posto n° 1
Prenotato posto n° 2
...
Prenotato posto n° 99
Prenotato posto n° 100
Problema con la prenotazione: posti esauriti!
Prenotato posto n° 101
```

Questa è una contraddizione! Gestire eccezioni è sempre un'operazione da compiere, ma non sempre bisogna gestire eccezioni laddove si presentano. In questo caso, l'ideale sarebbe gestire l'eccezione nella classe `GestorePrenotazioni`, piuttosto che nella classe `Botteghino`:

```
public class GestorePrenotazioni {
    public static void main(String [] args) {
        Botteghino botteghino = new Botteghino();
        try {
            for (int i = 1; i <= 101; ++i){
                botteghino.prenota();
                System.out.println("Prenotato posto n° " +
i);
            }
        }
        catch (PrenotazioneException exc) {
            System.out.println(exc.toString());
        }
    }
}
```

Tutto ciò è fattibile grazie al meccanismo di propagazione dell'eccezione di Java. Per compilare la classe `Botteghino` però, non basta rimuovere il blocco `try-catch` dal metodo `prenota()`, ma bisogna anche utilizzare la parola chiave `throws` nel seguente modo:

```
public void prenota() throws PrenotazioneException {
    // controllo sulla disponibilità dei posti
```

```
    if (postiDisponibili == 0) {
        //lancio dell'eccezione
        throw new PrenotazioneException();
    }
    // metodo che realizza la prenotazione
    // se non viene lanciata l'eccezione
    postiDisponibili--;
}
```

In questo modo otterremo il seguente desiderabile output:

```
Prenotato posto n° 1
Prenotato posto n° 2
...
Prenotato posto n°99
Prenotato posto n°100
Problema con la prenotazione: posti esauriti!
```

Se non utilizzassimo la clausola `throws` nella dichiarazione del metodo, il compilatore non compilerebbe il codice precedente. Segnalerebbe che il metodo `prenota()` potrebbe lanciare l'eccezione `PrenotazioneException` (che è evidente al compilatore per la presenza della parola chiave `throw`) e che questa non viene gestita. In particolare il messaggio di errore restituito sarebbe simile al seguente:

```
Botteghino.java:5: unreported exception PrenotazioneException;
  must be caught or declared to be thrown
```

Questo messaggio è una ulteriore prova delle caratteristiche di robustezza di Java. Con la clausola `throws` nella dichiarazione del metodo è come se avvertissimo il compilatore che siamo consapevoli che il metodo può lanciare al runtime la `PrenotazioneException` e di non preoccuparsi, perché gestiremo in un'altra parte del codice l'eccezione.

Se un ipotetico metodo `chiamante` volesse utilizzare il metodo `daChiamare`, che dichiara con una clausola `throws` il possibile lancio di un certo tipo di eccezione, il metodo `chiamante` dovrebbe gestire l'eccezione con un blocco

`try-catch` che include la chiamata al metodo `daChiamare`, oppure dovrebbe dichiarare anch'esso una clausola `throws` alla stessa eccezione. Ad esempio, questo vale per il metodo `main()` della classe `GestorePrenotazioni`.

Molti metodi della libreria standard sono dichiarati con clausola `throws` di qualche eccezione. Per esempio numerosi metodi delle classi del package `java.io` dichiarano clausole `throws` alla `IOException` (eccezione di input - output). Appare ancora più chiara ora la categorizzazione tra eccezioni checked ed unchecked: le checked exception devono essere per forza gestite per poter compilare, mentre le unchecked no, dato che si presentano solo al runtime.

È possibile dichiarare nella clausola `throws` anche più di un'eccezione, separando le varie tipologie con virgole, come nel seguente esempio:

```
public void prenota() throws PrenotazioneException,
NullPointerException {
    //...
}
```

Prima dell'avvento della versione 7, il seguente codice:

```
    class FirstException extends
Exception { }
    class SecondException extends
Exception { }
    public void rethrowException(String
exceptionName)
      throws FirstException,
SecondException {
        try {
            if
(exceptionName.equals("First")) {
```

```
                throw new
FirstException();
            } else {
                throw new
SecondException();
            }
        } catch (Exception e) {
            throw e;
        }
}
```

non era compilabile. Nonostante fosse evidente che il metodo in questione rilanciasse una delle due eccezioni `FirstException` **o** `SecondException`, **avendo utilizzato un blocco catch che rilancia una** `Exception`, **bisognava sostituire (o quantomeno aggiungere) nella clausola** `throws` **anche** `Exception`. **Java 7 ha eliminato questo comportamento introducendo un'analisi a livello di compilazione che verifica la veridicità della clausola** `throws`.

Sin dalla versione 1.4 del linguaggio fu stata introdotta una nuova caratteristica alle eccezioni. La classe `Throwable` fu modificata per supportare un semplice meccanismo di wrapping. Spesso infatti, si rende necessario per lo sviluppatore catturare una certa eccezione per lanciarne un'altra. Per esempio:

```
try {
    //...
} catch(UnaCertaEccezione e) {
    throw new UnAltraEccezione();
}
```

In tali casi, però, l'informazione della prima eccezione (nell'esempio `UnaCertaEccezione`) viene persa. Si era quindi costretti a creare un'eccezione personalizzata che poteva contenerne un'altra come variabile d'istanza. Per esempio:

```
public class WrapperException extends Exception {
    private Exception altraEccezione;
    public WrapperException(Exception altraEccezione) {
        this.setAltraEccezione(altraEccezione);
    }
    public void setAltraEccezione(Exception altraEccezione)
{
        this.altraEccezione = altraEccezione;
    }
    public Exception getAltraEccezione() {
        return altraEccezione;
    }
    //...
}
```

Con questo tipo di eccezione è possibile includere un'eccezione in un'altra, così:

```
try {
    //...
} catch(UnaCertaEccezione e) {
    throw new WrapperException(e);
}
```

Ma dalla versione 1.4 non bisogna più creare un'eccezione personalizzata per ottenere eccezioni wrapper. Nella classe `Throwable` sono stati introdotti i metodi `getClause()` e `initClause(Throwable)` e due nuovi costruttori, `Throwable(Throwable)` e `Throwable(String, Throwable)`. È quindi ora possibile per esempio codificare le seguenti istruzioni:

```
try {
    //...
} catch(ArithmeticException e) {
    throw new SecurityException(e);
}
```

È inoltre possibile concatenare un numero arbitrario di eccezioni.

9.5.1 Warnings

Quando compiliamo un file Java, il compilatore oltre a compilare correttamente o dare errori, potrebbe anche restituire dei warning (in italiano

“avvertimento”). Per esempio, compilando la classe `PrenotazioneException` (e in generale una qualsiasi classe che estende `Throwable`) con un IDE qualsiasi (Eclipse, Netbeans o anche EJE), otterremo il seguente output:

```
PrenotazioneException.java:1: warning: [serial] serializable
class
  PrenotazioneException has no definition of serialVersionUID
public class PrenotazioneException extends Exception {
       ^
1 warning
```

Non si tratta di un errore, la classe è stata compilata e il file .class è stato creato. Il compilatore ci sta solo *avvertendo* che questo file è stato definito in modo potenzialmente pericoloso. In particolare ci è stato segnalato che la nostra classe è serializzabile e non ha definito un `serialVersionUID`. Arrivati a questo punto del libro non abbiamo ancora le nozioni necessarie a comprendere esattamente il significato di questo warning. Tuttavia una spiegazione almeno parziale è dovuta, anche se probabilmente sarà tutto più chiaro dopo aver affrontato l’argomento della serializzazione. La classe `Throwable` implementa un’interfaccia denominata `Serializable` (in italiano “serializzabile”). Questa è definita nel package `java.io` della libreria standard. È un’interfaccia che non dichiara metodi, ma che se implementata permette agli oggetti che la implementano di essere *serializzati*. Un oggetto serializzabile per esempio può essere immagazzinato in un file, o fatto viaggiare in rete. Ora non ci dilungheremo nei tecnicismi della serializzazione (che sarà affrontata approfonditamente nel capitolo 18 relativo all’Input-Output), visto che il nostro obiettivo è solo quello di capire il significato di questo warning, basti sapere che se una classe è serializzabile ci viene chiesto dal compilatore di definire un attributo denominato `serialVersionUID`. Questo potrebbe rendersi necessario nel caso la nostra classe dovesse essere usata in ambienti distribuiti (ovvero nel caso gli oggetti della classe dovessero essere usati da virtual machine diverse viaggiando attraverso la rete). In casi come questi dovrebbe esserci una copia della classe serializzabile su macchine diverse. Questo comporta che senza la dichiarazione esplicita del `serialVersionUID` a garantire che le due versioni siano compatibili, la lettura dell’oggetto serializzabile potrebbe fallire e potrebbe essere lanciata una `InvalidClassException`. Per risolvere questo

warning è possibile dichiarare il `serialVersionUID` per esempio nel seguente modo:

```
private static final long serialVersionUID =
8144963013726442881L;
```

Si noti che è possibile specificare un qualsiasi modificatore d'accesso, anche se solitamente si preferisce `private`, visto che non c'è ragione di esporre all'esterno questa attributo. È obbligatorio invece che si tratti di un long dichiarato `static` e `final`. Il valore da assegnare a `serialVersionUID` è arbitrario, sarebbe bene generarne uno univoco come nell'esempio, ma anche il valore `1L` andrebbe bene! Gli IDE come Eclipse e Netbeans posseggono delle facilities che permettono la generazione automatica di questi valori.

Un'altra possibilità per risolvere il warning, ma valida solo nel caso si sia sicuri che questa classe non debba essere serializzata, è quella di usare l'annotazione `SuppressWarning` sulla classe stessa nel seguente modo:

```
@SuppressWarnings("serial")
public class PrenotazioneException extends Exception {
```

In questo caso è come se avessimo raccomandato al compilatore di non preoccuparsi perché il rischio è controllato. Nel caso invece compiliate direttamente da riga di comando nel seguente modo:

```
javac PrenotazioneException.java
```

nessun warning sarà lanciato. Infatti, affinché vengano segnalati i nostri warning, bisogna compilare con l'opzione -Xlint:all (o semplicemente con -Xlint):

```
javac -Xlint:all PrenotazioneException.java
```

è anche possibile specificare quale tipo di warning deve essere segnalato, per esempio con il seguente comando:

```
javac -Xlint:serial PrenotazioneException.java
```

saranno segnalati solo warning sul `serialVersionUID` mancante.

Per maggiori informazioni sulle opzioni del compilatore, è possibile consultare la documentazione dei tool del JDK.

In generale anche dai vari IDE dovrebbe essere possibile impostare un'opzione affinché il compilatore non segnali dei warning.

> **Per quanto riguarda EJE bisogna andare nel menu File⇨ Opzioni⇨Tab Java e deselezionare il check box relativo ai warning.**

9.5.2 Altra regola sull'override

Nonostante molti sviluppatori considerino inutili le checked exception ed evitino accuratamente di utilizzarle nei propri programmi, i progettisti di Java le hanno concepite pensandole come una parte dell'interfaccia di un metodo. Infatti quando abbiamo parlato di override di un metodo nel capitolo precedente, abbiamo elencate tre **regole.**

1. Il metodo riscritto nella sottoclasse deve avere la stessa firma (nome e parametri) del metodo della superclasse.
2. Il tipo di ritorno del metodo della sottoclasse deve coincidere con quello del metodo che si sta riscrivendo, o deve essere di un tipo che estende il tipo di ritorno del metodo della superclasse.
3. Il metodo ridefinito nella sottoclasse non deve essere meno accessibile del metodo originale della superclasse.

Adesso dobbiamo introdurne un'altra:

4. Quando si fa override di un metodo non è possibile specificare clausole `throws` su eccezioni che il metodo base non comprende nella propria clausola `throws`. È comunque possibile da parte del metodo che effettua override dichiarare una clausola `throws` per eccezioni che sono sottotipi di eccezioni comprese dal metodo base nella propria clausola `throws`.

Cerchiamo di spiegare questa regola sinteticamente con un esempio:

```
public class ClasseBase {
```

```
    public void metodo() throws java.io.IOException {}
}

public class SottoClasseCorretta1 extends ClasseBase {
    public void metodo() throws java.io.IOException {}
}

public class SottoClasseCorretta2 extends ClasseBase {
    public void metodo() throws
java.io.FileNotFoundException {}
}

public class SottoClasseCorretta3 extends ClasseBase {
    public void metodo() {}
}

public class SottoClasseScorretta extends ClasseBase {
    public void metodo() throws java.sql.SQLException {}
}
```

La classe `ClasseBase` comprende un metodo che dichiara nella sua clausola `throws` una `IOException`. La classe `SottoClasseCorretta1` effettua override del metodo e dichiara la stessa `IOException` nella sua clausola `throws`. La classe `SottoClasseCorretta2` fa override del metodo e dichiara una `FileNotFoundException`, che è sottoclasse di `IOException` nella sua clausola `throws`. La classe `SottoClasseCorretta3` fa override del metodo e non dichiara clausole `throws`. Infine la classe `SottoClasseScorretta` fa override del metodo e dichiara una `SQLException` nella sua clausola `throws`, e ciò è illegale.

9.6 Introduzione alle asserzioni

Quando nella versione 1.4 di Java furono introdotte le asserzioni, per la prima volta si modificò la lista delle parole chiave con l'aggiunta di una nuova: `assert`, cosa mai accaduta nelle precedenti major release.

Un'**asserzione** è un'istruzione che consente di testare eventuali comportamenti previsti in un'applicazione. Ogni asserzione richiede sia verificata un'espressione booleana che lo sviluppatore ritiene vada verificata nel punto in cui viene dichiarata. Se la verifica è negativa si deve parlare di un errore nell'implementazione del codice, oppure in gergo, di *bug*. Le

asserzioni possono quindi rappresentare un'utile strumento per accertarsi che il codice scritto si comporti come ci si aspetta. Lo sviluppatore può disseminare il codice di asserzioni in modo da testare la robustezza del codice in maniera semplice ed efficace. Lo sviluppatore può infine disabilitare la lettura delle asserzioni da parte della JVM in fase di rilascio del software, così che l'esecuzione non sia rallentata. Moltissimi sviluppatori pensano che l'utilizzo delle asserzioni sia una delle tecniche di maggior successo per scovare bug. Eppure le asserzioni sono relativamente poco usate nell'ambito della programmazione Java. Forse per qualcuno sono fuori moda, ma la realtà è che ai più non è molto chiaro come debbano essere utilizzate. Inoltre, le asserzioni rappresentano anche un ottimo strumento per documentare il comportamento interno di un programma, favorendo la manutenibilità dello stesso.

Esistono due tipi di sintassi per utilizzare le asserzioni:

1. `assert espressione_booleana;`
2. `assert espressione_booleana: espressione_stampabile;`

Con la sintassi 1), quando l'applicazione esegue l'asserzione valuta `espressione_booleana`. Se il valore dell'espressione è `true` il programma prosegue normalmente, ma se il valore è `false` viene lanciato l'errore `AssertionError`. Per esempio l'istruzione:

```
assert b > 0;
```

è semanticamente equivalente a:

```
if (!(b>0)) {
    throw new AssertionError();
}
```

A parte l'eleganza e la compattezza del costrutto `assert`, la differenza tra le due precedenti espressioni è notevole. Le asserzioni rappresentano, più che un'istruzione, uno strumento per testare la veridicità delle assunzioni formulate dallo sviluppatore sulla propria applicazione. Se la condizione che viene asserita dal programmatore è falsa, l'applicazione terminerà immediatamente mostrando le ragioni tramite uno stack-trace (metodo `printStackTrace()` di cui sopra). Infatti si è verificato qualcosa che non era

previsto dallo sviluppatore stesso. È possibile disabilitare la lettura delle asserzioni da parte della JVM una volta rilasciato il proprio prodotto, al fine di non rallentarne l'esecuzione. Ciò evidenzia la differenza tra le asserzioni e tutte le altre istruzioni applicative: le asserzioni possono essere disabilitate, mentre le altre istruzioni no.

Rispetto alla sintassi 1), la sintassi 2) consente di specificare anche un messaggio esplicativo tramite `espressione_stampabile`. Per esempio:

```
assert b > 0: b;
```

oppure:

```
assert b > 0: "il valore di b è " + b;
```

o:

```
assert b > 0: getMessage();
```

o anche:

```
assert b > 0: "assert b > 0 = " + (b > 0);
```

`espressione_stampabile` può essere una qualsiasi espressione che ritorni un qualche valore (quindi non è possibile invocare un metodo con tipo di ritorno `void`). La sintassi 2) permette dunque di migliorare lo stack-trace delle asserzioni.

Il tool JUnit è probabilmente il più famoso software per eseguire test automatici. Tra gli autori figurano due mostri sacri dell'informatica come Kent Beck, autore del processo Test Driven Development e della metodologia Extreme Programming, ed Erich Gamma, storico co-autore del primo libro sui Design Pattern. In particolare JUnit è stato creato per creare "unit test" (in italiano "test unitari"), ovvero creare software che testa automaticamente ognuna delle classi che creiamo. Il grande vantaggio di questo approccio è che alla fine di una giornata lavorativa (ma anche all'inizio o durante), sarà possibile testare la

correttezza delle nostre classi con un semplice click, ed ottenere un eventuale report dei bachi di regressioni del nostro progetto creati con lo sviluppo giornaliero. JUnit è completamente basato su asserzioni, ed esiste da prima che le asserzioni fossero introdotte in Java. Infatti gli autori crearono dei metodi, come `assertEquals()` **o** `assertTrue()`**, per verificare la correttezza del codice. Per maggiori informazioni su JUnit, consultare l'indirizzo:** http://junit.org/.

9.6.1 Progettazione per contratto

Il meccanismo delle asserzioni deve il suo successo ad una tecnica di progettazione nota con il nome di **Progettazione per contratto (Design by contract)**, sviluppata da Bertrand Meyer. Tale tecnica è una caratteristica fondamentale del linguaggio di programmazione sviluppato da Meyer stesso: Eiffel (per informazioni http://www.eiffel.com). Ma è possibile progettare per contratto, più o meno agevolmente, con qualsiasi linguaggio di programmazione. La tecnica si basa in particolare su tre tipologie di asserzioni: precondizioni, postcondizioni ed invarianti (le invarianti a loro volta si dividono in interne, di classe, sul flusso di esecuzione etc.).

Con una **precondizione** lo sviluppatore può specificare quale deve essere lo stato dell'applicazione nel momento in cui viene invocata un'operazione. In questo modo si rende esplicito chi ha la responsabilità di testare la correttezza dei dati. L'utilizzo dell'asserzione riduce sia il pericolo di dimenticare completamente il controllo, sia quello di effettuare troppi controlli (perché si possono abilitare e disabilitare). Dal momento che si tende ad utilizzare le asserzioni in fase di test e debugging, non bisogna mai confondere l'utilizzo delle asserzioni con quello della gestione delle eccezioni. Nel paragrafo 9.6.3 verranno esplicitate regole da seguire per l'utilizzo delle asserzioni.

Con una **postcondizione** lo sviluppatore può specificare quale deve essere lo stato dell'applicazione nel momento in cui un'operazione viene completata. Le postcondizioni rappresentano un modo utile per dire che cosa fare, senza dire come. In altre parole, è un altro metodo per separare

interfaccia ed implementazione interna.

È infine utilizzabile il concetto di **invariante**, che se applicato ad una classe, permette di specificare vincoli per tutti gli oggetti istanziati. Questi possono trovarsi in uno stato che non rispetta il vincolo specificato (detto *stato inconsistente)* solo temporaneamente durante l'esecuzione di qualche metodo, al termine del quale lo stato deve ritornare *consistente*.

La progettazione per contratto è appunto una tecnica di progettazione e non di programmazione. Essa permette per esempio di testare la consistenza dell'ereditarietà. Una sottoclasse, infatti, potrebbe indebolire le precondizioni e fortificare postcondizioni e invarianti di classe al fine di convalidare l'estensione. Al lettore interessato ad approfondire le sue conoscenze sulla progettazione per contratto, consigliamo di dare uno sguardo alla bibliografia.

> **Se la vostra esigenza fosse quella di usare codice preesistente creato con una versione di Java precedente alla versione 1.4, dovreste consultare l'appendice U.**

9.6.2 Note per l'esecuzione di programmi che utilizzano la parola `assert`

Come più volte detto, è possibile in fase di esecuzione abilitare o disabilitare le asserzioni. Come al solito bisogna utilizzare flag, questa volta applicandoli al comando java, ovvero -enableassertions (o più brevemente -ea) per abilitare le asserzioni, e -disableassertions (o -da) per disabilitarle. Per esempio:

```
java -ea MioProgrammaConAsserzioni
```

abilita da parte della JVM la lettura dei costrutti `assert`. Mentre:

```
java -da MioProgrammaConAsserzioni
```

disabilita le asserzioni per non rallentare l'applicazione. Siccome le asserzioni sono disabilitate di default, il comando precedente è equivalente al seguente:

```
java MioProgrammaConAsserzioni
```

Sia per l'abilitazione sia per la disabilitazione valgono le seguenti regole:

1. se non si specificano argomenti dopo i flag di abilitazione o disabilitazione delle asserzioni, saranno abilitate o disabilitate le asserzioni in tutte le classi del nostro programma (ma non nelle classi della libreria standard utilizzate). Questo è il caso dei precedenti esempi.
2. Specificando il nome di un package seguito da tre puntini, si abilitano o disabilitano le asserzioni in quel package e in tutti i sottopackage. Per esempio il comando:

```
java -ea -da:miopackage... MioProgramma
```

   abiliterà le asserzioni in tutte le classi tranne quelle del package `miopackage`; specificando solo i tre puntini, si abilitano o disabilitano le asserzioni nel package di default (ovvero la cartella da dove parte il comando).

3. Specificando solo il nome di una classe, si abilitano o disabilitano le asserzioni in quella classe. Per esempio il comando:

```
java -ea:... -da:MiaClasse MioProgramma
```

   abiliterà le asserzioni in tutte le classi del package di default, tranne che nella classe `MiaClasse`.

È anche possibile abilitare o disabilitare le asserzioni delle classi della libreria standard che si vuole utilizzare mediante i flag -enablesystemassertions (o più brevemente -esa) e -disablesystemassertions (o -dsa). Anche per questi flag valgono le regole di cui sopra.

Per quanto riguarda la fase di esecuzione non esistono differenze sul come sfruttare le asserzioni tra la versione 1.4 e le versioni successive.

In alcuni programmi critici è possibile che lo sviluppatore si voglia assicurare che le asserzioni siano abilitate. Con il seguente blocco di codice statico:

```
static {
```

```
    boolean assertsEnabled = false;
    assert assertsEnabled = true;
    if (!assertsEnabled)
       throw new
RuntimeException("Asserts must be " +
                                 "enabled!");

}
```

è possibile garantire che il programma sia eseguibile solo se le asserzioni sono abilitate. Il blocco, infatti, prima dichiara ed inizializza la variabile booleana `assertsEnabled` **a** `false`**, per poi cambiare il suo valore a** `true` **se le asserzioni sono abilitate. Se le asserzioni non sono abilitate, il programma termina con il lancio della** `RuntimeException`**, altrimenti continua. Ricordiamo che il blocco statico viene eseguito (cfr. capitolo 6) un'unica volta nel momento in cui la classe che lo contiene viene caricata. Per questa ragione il blocco statico dovrebbe essere inserito nella classe del** `main()` **per essere sicuri di ottenere il risultato voluto.**

9.6.3 Quando usare le asserzioni

Non tutti gli sviluppatori possono essere interessati all'utilizzo delle asserzioni. Un'asserzione non può ridursi a un modo conciso di esprimere una condizione regolare. Un'asserzione è invece il concetto fondamentale di una metodologia di progettazione volta a rendere i programmi più robusti. Nel momento in cui però lo sviluppatore decide di utilizzare tale strumento, dovrebbe essere suo interesse utilizzarlo correttamente. I seguenti consigli derivano dall'esperienza e dallo studio dell'autore sui testi relativi alle asserzioni.

9.6.3.1 Consiglio 1

È spesso consigliato (anche nella documentazione ufficiale) di non utilizzare

precondizioni per testare la correttezza dei parametri di metodi pubblici. È invece raccomandato l'utilizzo delle precondizioni per testare la correttezza dei parametri di metodi privati, protetti o con visibilità a livello di package. Questo dipende dal fatto che un metodo non pubblico ha la possibilità di essere chiamato da un contesto limitato, noto, corretto e funzionante. Ciò implica la presunzione che le nostre chiamate al metodo in questione siano corrette, ed è quindi lecito rinforzare tale concetto con un'asserzione. Supponiamo di avere un metodo con visibilità di package come il seguente:

```
public class InstancesFactory {
    Object getInstance(int index) {
        assert (index == 1 || index == 2);
        switch (index) {
            case 1:
                return new Instance1();
            case 2:
                return new Instance2();
        }
    }
}
```

La classe `InstancesFactory` implementa una soluzione personalizzata basata sul pattern denominato **Factory Method** (per informazioni sul concetto di pattern, cfr. appendice D). Se questo metodo può essere chiamato solo da classi che appartengono allo stesso package della classe `InstancesFactory`, non deve mai accadere che il parametro `index` sia diverso da `1` o `2`, perché tale situazione rappresenterebbe un bug.

Se invece il metodo `getInstance()`, fosse dichiarato `public`, la situazione sarebbe diversa. Infatti, un eventuale controllo del parametro `index` dovrebbe essere considerato ordinario, e quindi da gestire magari mediante il lancio di un'eccezione:

```
public class InstancesFactory {
    public Object getInstance(int index) throws Exception {
        if (!(index == 1 || index == 2)) {
            throw new Exception("Indice errato: " + index);
        }
        switch (index) {
            case 1:
                return new Instance1();
```

```
            case 2:
                return new Instance2();
        }
    }
}
```

L’uso di un’asserzione, in tal caso, non garantirebbe la robustezza del programma, ma solo la sua eventuale interruzione se fossero abilitate le asserzioni al runtime, non potendo controllare a priori che il metodo sia invocato nel modo corretto. Di conseguenza una precondizione di questo tipo violerebbe il concetto object oriented di metodo pubblico.

9.6.3.2 Consiglio 2

È sconsigliato l’utilizzo di asserzioni laddove si vuole testare la correttezza di dati che sono inseriti da un utente. Le asserzioni dovrebbero testare la consistenza del programma con sé stesso, non la consistenza dell’utente con il programma. L’eventuale input non corretto da parte di un utente è giusto che sia gestito mediante eccezioni, non asserzioni. Per esempio, modifichiamo la classe `Data` di cui abbiamo parlato nel capitolo 6 per spiegare l’incapsulamento:

```
public class Data {
    private int giorno;
    //...
    public void setGiorno(int g) {
        assert (g > 0 && g <= 31): "Giorno non valido";
        giorno = g;
    }
//...
```

dove il parametro `g` del metodo `setGiorno()` veniva passato da un utente mediante un oggetto interfaccia, che rappresentava un’interfaccia grafica (codice 6.2 bis):

```
//...
Data unaData = new Data();
unaData.setGiorno(interfaccia.dammiGiornoInserito());
unaData.setMese(interfaccia.dammiMeseInserito());
unaData.setAnno(interfaccia.dammiAnnoInserito());
//...
```

Come il lettore avrà intuito, l'utilizzo della parola chiave `assert` non è corretto. Infatti, nel caso le asserzioni fossero abilitate in fase di esecuzione dell'applicazione e l'utente inserisse un valore errato per inizializzare la variabile giorno, l'applicazione si interromperebbe con un `AssertError!` Se le asserzioni non fossero abilitate, nessun controllo impedirebbe all'utente di inserire valori errati. La soluzione ideale sarebbe gestire la situazione tramite un'eccezione, per esempio:

```
public void setGiorno(int g) throws RuntimeException {
    if (!(g > 0 && g <= 31)) {
        throw new RuntimeException("Giorno non valido");
    }
    giorno = g;
}
```

Naturalmente la condizione è ampiamente migliorabile.

9.6.3.3 Consiglio 3

L'uso delle asserzioni invece, ben si adatta alle postcondizioni ed alle invarianti. Per **postcondizione** intendiamo una condizione che viene verificata appena prima che termini l'esecuzione di un metodo (ultima istruzione). Segue un esempio:

```
public class Connection {
    private boolean isOpen = false;
    public void open() {
        //...
        isOpen = true;
        //...
        assert isOpen;
    }
    public void close() throws ConnectionException {
        if (!isOpen) {
            throw new ConnectionException(
              "Impossibile chiudere connessioni non
aperte!" );
        }
        //...
        isOpen = false;
        //...
        assert !isOpen;
```

```
    }
}
```

Dividiamo le **invarianti in interne, di classe** e **sul flusso di esecuzione.** Per **invarianti interne** intendiamo asserzioni che testano la correttezza dei flussi del nostro codice. Per esempio il seguente blocco di codice:

```
if (i == 0) {
    //...
} else if (i == 1) {
    //...
} else { // sicuramente vale per (i == 2)
    //...
}
```

può diventare più robusto con l'uso di un'asserzione:

```
if (i == 0) {
    //...
} else if (i == 1) {
    //...
} else {
    assert i == 2 :  "Attenzione i = " + i + "!";
    //...
}
```

Un tale tipo di invariante viene usato con più probabilità all'interno di una clausola `default` di un costrutto `switch`. Spesso lo sviluppatore sottovaluta il costrutto omettendo la clausola `default`, perché suppone che il flusso passi sicuramente per un certo `case`. Le asserzioni sono molto utili per convalidare le nostre supposizioni. Per esempio, il seguente blocco di codice:

```
switch(tipoAuto) {
    case Auto.SPORTIVA:
        //...
    break;
    case Auto.LUSSO:
        //...
    break;
    case Auto.UTILITARIA:
        //...
    break;
}
```

può diventare più robusto con l'uso di un'asserzione:

```
switch(tipoAuto) {
    case Auto.SPORTIVA:
        //...
    break;
    case Auto.LUSSO:
        //...

    break;
    case Auto.UTILITARIA:
        //...
    break;
    default:
        assert false :  "Tipo auto non previsto :  " +
        tipoAuto;
}
```

Per **invarianti di classe** intendiamo particolari invarianti interne che devono essere vere per tutte le istanze di una certa classe, in ogni momento del loro ciclo di vita, tranne che durante l'esecuzione di alcuni metodi. All'inizio ed al termine di ogni metodo, però, lo stato dell'oggetto deve tornare *consistente*. Per esempio, un oggetto della seguente classe:

```
public class Bilancia {
    private double peso;
    public Bilancia() {
        azzeraLancetta();
        assert lancettaAzzerata(); // invariante di classe
    }
    private void setPeso(double grammi) {
        assert grammi >= 0; // pre-condizione
        peso = grammi;
    }
    private double getPeso() {
        return peso;
    }
    public void pesa(double grammi) {
        if (grammi <= 0) {
            throw new RuntimeException("Grammi <= 0! ");
        }
        setPeso(grammi);
        mostraPeso();
        azzeraLancetta();
```

```
        assert lancettaAzzerata(); // invariante di classe
    }
    private void mostraPeso() {
        System.out.println("Il peso è di " + peso + "
grammi");
    }
    private void azzeraLancetta() {
        setPeso(0);
    }
    private boolean lancettaAzzerata () {
        return peso == 0;
    }
}
```

potrebbe, dopo ogni pesatura, azzerare la lancetta (si noti che solo i due metodi pubblici terminano con un'asserzione).

Per **invarianti sul flusso di esecuzione** intendiamo asserzioni posizionate in zone del codice che non dovrebbero mai essere raggiunte. Per esempio, se abbiamo codice commentato in questo modo:

```
public void metodo() {
    if (flag == true) {
        return;
    }
    // L'esecuzione non dovrebbe mai arrivare qui!
}
```

Potremmo sostituire il commento con un'asserzione `false`:

```
public void metodo() {
    if (flag == true) {
        return;
    }
    assert false;
}
```

9.6.4 Conclusioni

In questo capitolo abbiamo raggruppato argomenti che possono sembrare simili, e ne abbiamo esplicitato le differenze. Il concetto di fondo è che l'utilizzo della gestione delle eccezioni è fondamentale per la creazione di applicazioni robuste. Le asserzioni invece rappresentano un comodo

meccanismo per testare la robustezza delle nostre applicazioni. La gestione delle eccezioni non è consigliabile, è obbligatoria! Le asserzioni invece sono molto spesso ingiustamente snobbate. In effetti bisogna avere un po' di esperienza per sfruttarne a pieno le potenzialità. Infatti, la progettazione per contratto è un argomento complesso che va studiato a fondo per poter ottenere risultati migliori. Ciononostante anche l'utilizzo dei concetti più semplici come le postcondizioni può migliorare la robustezza delle nostre applicazioni.

Riepilogo

In questo capitolo abbiamo dapprima distinto i concetti di eccezione, errore ed asserzione. La **gerarchia delle classi** che possono essere lanciate in Java parte dalla classe **`Throwable`**, la quale è estesa da **`Error`** ed **`Exception`**. Abbiamo categorizzato le eccezioni e gli errori con una panoramica sulle classi principali. Inoltre abbiamo ulteriormente suddiviso le tipologie di eccezioni in **unchecked** (tutte le sottoclassi di `RuntimeException`) e **checked** (tutte le altre eccezioni). È stato presentato il meccanismo alla base della gestione delle eccezioni, parallelamente alle cinque parole chiave che ne consentono la gestione. I blocchi **`try-catch-finally`** permettono di gestire localmente le eccezioni. Le coppie **`throw-throws`** supportano invece la propagazione (in maniera robusta) delle eccezioni.

Estendendo la classe `Exception` (o le sue sottoclassi), possiamo creare le nostre eccezioni personalizzate. Queste solitamente devono essere lanciate da noi con il comando `throw` (e se si tratta di checked exception devono anche essere dichiarate con il comando `throws`).

La possibilità di astrarre il concetto di eccezione con gli oggetti e la possibilità di sfruttare il meccanismo di call-stack (**propagazione dell'errore-eccezione**) permettono di creare applicazioni contemporaneamente object-oriented, semplici e robuste, senza doverci preoccupare di progettare un tipo di ritorno che contempli anche la possibilità di specificare un problema. Il costrutto **try with resources** ci garantisce semplicità e robustezza nei casi in cui abbiamo a che fare con oggetti che implementano l'interfaccia `Closeable` o `AutoCloseable`.

Le asserzioni possono essere dichiarate in maniera sintetica o

specificando anche un eventuale messaggio di errore. Hanno la caratteristica di potere essere abilitate o disabilitate al momento dell'esecuzione del programma, e sono fornite di diverse opzioni di utilizzo da specificare da riga di comando.

Abbiamo anche introdotto il loro utilizzo all'interno della progettazione per contratto, introducendo i concetti di precondizioni, postcondizioni ed invarianti. Le **precondizioni** vengono utilizzate all'inizio di un metodo, e servono per testare l'input nel caso il metodo non sia pubblico. Le **postcondizioni** invece sono posizionate all'uscita del metodo, e servono per testare la correttezza dei risultati del metodo. Le **invarianti** invece possono trovarsi all'interno del codice di un metodo e si dividono in invarianti di classe, interne e sul flusso di esecuzione. Per **invarianti interne** intendiamo asserzioni che testano la correttezza dei flussi del nostro codice. Si trovano spesso all'interno di algoritmi per testare situazioni intermedie. Per **invarianti di classe** intendiamo particolari invarianti interne che devono essere vere per tutte le istanze di una certa classe, in ogni momento del loro ciclo di vita, tranne che durante l'esecuzione di alcuni metodi. Per **invarianti sul flusso di esecuzione** intendiamo asserzioni posizionate in zone del codice che non dovrebbero mai essere raggiunte.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Conclusioni Parte II

Qui termina la seconda parte del libro. Mentre nella prima parte sono stati presentati quasi esclusivamente argomenti di programmazione, in questa seconda parte abbiamo cercato di concentrarci sulla parte teorica ragionando anche su analisi e progettazione. L'obiettivo è quello di indirizzare il lettore verso pratiche che lo portino a creare codice di qualità, facile da modificare e con cui è facile interagire. Come è facile immaginare la pratica sarà la migliore alleata. Infatti solo affrontando in prima persona i problemi descritti, ci si renderà conto di quanto possano influire davvero sul fatto che un software si possa considerare un successo o meno.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere le differenze tra eccezioni, errori ed asserzioni (unità 9.1)	☐	
Saper gestire le varie tipologie di eccezioni con i blocchi `try-catch` (unità 9.2, 9.3)	☐	
Saper creare tipi di eccezioni personalizzate e gestire il meccanismo di propagazione con le parole chiave `throw` e `throws` (unità 9.2, 9.5)	☐	
Saper utilizzare il costrutto *try with resources* (unità 9.4)	☐	
Capire e saper utilizzare le asserzioni (unità 9.6)	☐	

Note:

Parte III

Caratteristiche avanzate

La parte III presenterà al lettore tutte le caratteristiche avanzate del linguaggio. L'approfondimento di alcuni concetti è molto elevato, e in generale i concetti stessi sono mediamente più complicati di quelli visti sino ad ora. Anche argomenti apparentemente semplici come le enumerazioni, presentano situazioni piuttosto complesse. Nei prossimi moduli esploreremo nuovi tipi di programmazione, come quella generica, quella concorrente (multithreaded) e quella funzionale grazie alle espressioni lambda. Inoltre introdurremo tutte le librerie fondamentali come la libreria Date and Time API, le librerie per gestire input e output, il framework Collections, la libreria Stream API e tante altre. L'ultimo capitolo è dedicato alla novità più importante di Java 9: i moduli. La modularizzazione della piattaforma Java rivoluziona completamente il nostro modo di concepire le applicazioni Java, e permette di creare sottosistemi disaccoppiati riutilizzabili, ed incapsulare risorse inaccessibili all'esterno con il cosiddetto forte incapsulamento. Inoltre con il nuovo strumento jlink, è ora possibile creare runtime personalizzati che includono solamente le librerie della piattaforma Java che sono necessarie.

10

Enumerazioni e Tipi Innestati

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Comprendere cosa sono i tipi innestati e i vantaggi che comportano (unità 10.1).
- ✔ Saper elencare le proprietà fondamentali delle classi innestate (unità 10.1).
- ✔ Saper definire ed utilizzare le classi anonime (unità 10.2).
- ✔ Comprendere l'utilizzo e l'utilità delle enumerazioni (unità 10.3).
- ✔ Comprendere l'utilizzo e l'utilità delle caratteristiche avanzate delle enumerazioni (unità 10.3).

Nel capitolo 2 abbiamo introdotto i componenti fondamentali della programmazione Java, ma ci siamo limitati ai concetti indispensabili. Abbiamo solo accennato agli altri componenti fondamentali della programmazione Java nell'ultimo paragrafo visto che era prematuro approfondirne il discorso.

È ora arrivato il momento quindi di introdurre le enumerazioni, che al pari delle classi e delle interfacce (e come vedremo delle annotazioni), vengono solitamente salvate in file con suffisso .java. Il concetto di

enumerazione non è complicato, ma vi saranno da spiegare anche le caratteristiche avanzate.

In questo capitolo vedremo essenzialmente due argomenti principali: le classi innestate e le enumerazioni. La conoscenza delle classi innestate è propedeutica alla definizione avanzata delle enumerazioni. Entrambi gli argomenti sono abbastanza semplici da imparare se ci si limita all'utilizzo di base. Diventano molto più complessi quando si vogliono applicare le caratteristiche avanzate di queste nuove definizioni. Particolarmente ostiche sono le proprietà delle classi innestate, poco conosciute dalla maggior parte dei programmatori Java. Molto importanti dal punto di vista dell'utilizzo sono le classi anonime, che sino a Java 7 hanno rappresentato il modo più dinamico, sintetico e funzionale di scrivere codice Java. La seconda parte del capitolo è probabilmente più interessante. Infatti, definiremo le enumerazioni con il supporto di semplici esempi. Dopo aver presentato le ragioni per cui le enumerazioni costituiscono un vero punto di forza di Java, ne presenteremo le proprietà, le caratteristiche avanzate (metodi specifici, override tramite classi anonime, costruttori, etc.), il loro rapporto con i paradigmi object oriented, e con elementi di programmazione come il costrutto `switch` e gli import statici. Inoltre, cercheremo di capire che lavoro svolge il compilatore per noi, e noteremo come tutte le enumerazioni, inoltre, estendano la classe `java.lang.Enum` ereditandone i metodi.

10.1 Classi innestate: classi interne

Nel titolo di questo capitolo abbiamo parlato di **tipi innestati** perché lo stesso discorso che faremo in particolare per le classi, si potrà generalizzare anche per le interfacce, le enumerazioni e le annotazioni. Iniziamo a parlare delle **classi innestate** (in inglese **nested classes).**

10.1.1 Classe innestata: definizione

Una **classe innestata** (o anche **classe interna** nel caso non sia dichiarata statica) non è altro che una classe definita all'interno di un'altra classe. Per esempio:

```
public class Outer {
    private String messaggio = "Nella classe ";
```

```
    private void stampaMessaggio() {
        System.out.println(messaggio + "Esterna");
    }
    /* la classe interna accede ai membri privati
     * della classe che la contiene
    */
    public class Inner {
        public void metodo() {
            System.out.println(messaggio + "Interna");
        }
        public void chiamaMetodo() {
            stampaMessaggio();
        }
        //...
    }
    //...
}
```

Il vantaggio di implementare una classe all'interno di un'altra riguarda principalmente il risparmio di codice. Infatti la classe interna ha accesso ai membri della classe esterna anche se dichiarati `private`. Nell'esempio, il metodo `metodo()`, della classe interna `Inner` utilizza la variabile privata d'istanza `messaggio`, definita però nella classe esterna `Outer`. Stesso discorso per il metodo `chiamaMetodo()` della classe interna che chiama il metodo della classe esterna `stampaMessaggio()`. Dal punto di vista object oriented, di solito nessun vincolo o requisito dovrebbe consigliarci l'implementazione di una classe innestata. Questo significa che è sempre possibile evitarne l'utilizzo. Esistono dei casi dove però tali costrutti sono effettivamente molto comodi. Per esempio nella creazione di classi per la gestione degli eventi sulle interfacce grafiche, ma tale argomento sarà affrontato in dettaglio nell'appendice Q dedicata alle interfacce grafiche.

Anche per le classi innestate esiste la possibilità di utilizzare i modificatori d'accesso come per i membri di una classe. Quindi, per le classi ordinarie è possibile solo usare `public` oppure non usare modificatori, ed abbiamo visto nel capitolo 6 come questo implichi che quella classe sia accessibile al di fuori del package oppure no. Invece per le classi innestate è possibile anche utilizzare `protected` e `private`, con le stesse regole che esistono per gli attributi di una classe, anche se non si tratta di attributi.

Inoltre è possibile anche utilizzare altri modificatori come `static`. Nel

prossimo paragrafo vengono presentate le regole che governano le classi interne: sono complesse e difficili da ricordare.

10.1.2 Classi innestate: proprietà

Le seguenti proprietà vengono riportate per completezza. Tuttavia non ci dilungheremo sull'effettivo sfruttamento di queste regole. Dubitiamo fortemente per esempio che vengano usate spesso classi innestate statiche. Sicuramente però, il programmatore Java esperto può sfruttare con profitto tali proprietà per risolvere problemi di non facile soluzione. Una classe innestata:

1. deve avere un identificatore differente dalla classe che la contiene.
2. Si possono utilizzare tutti i modificatori d'accesso per dichiarare una classe innestata. Questo implica che gli oggetti di una classe innestata sono istanziabili solo negli ambiti di visibilità definiti dai modificatori come descritto nel paragrafo 6.7.
3. Per istanziare oggetti di una classe innestata (non privata) al di fuori della classe in cui è stata dichiarata, bisogna utilizzare una sintassi speciale:

`NomeClasseEsterna.nomeClasseInterna`. In particolare, se volessimo istanziare una classe interna al di fuori della classe in cui è definita, è necessario effettuare determinati passi. Facendo riferimento all'esempio precedente, bisogna:

i. istanziare la classe esterna (in cui è dichiarata la classe interna):

```
Outer outer = new Outer();
```

ii. dichiarare l'oggetto che si vuole istanziare dalla classe interna tramite la classe esterna (sfruttando l'operatore dot):

```
Outer.Inner inner;
```

iii. istanziare l'oggetto dalla classe interna tramite l'oggetto istanziato dalla classe esterna (si noti la chiamata all'operatore `new` invocato tramite l'operatore dot):

```
inner = outer.new Inner();
```

4. Ha accesso sia alle variabili d'istanza sia a quelle statiche della classe in cui è dichiarata.

5. Una classe innestata si può dichiarare anche all'interno di un metodo (o in generale in un blocco di codice come quello di un `if` o di un `for`) e in questo caso viene detta **classe locale** (in inglese **local class).** In questo caso le variabili locali saranno accessibili solo se dichiarate `final` oppure **effettivamente non modificat** e (in inglese **effectively final).** Per esempio il seguente codice:

```
public class Outer {
    private String stringaOuter = "JAVA";
    public void metodoOuter() {
        int intMetodo = 7;
        class Inner {
            public void metodoInner() {
                System.out.println(stringaOuter + " " +
(++intMetodo));
            }
        }
    }
}
```

produrrà un errore in compilazione perché l'espressione `++intMetodo` prova ad assegnare un nuovo valore ad una variabile non dichiarata nella classe interna. Sino a Java 7 era obbligatorio comunque dichiarare la variabile `final`. La regola è stata resa più flessibile per consentire un'introduzione più graduale delle espressioni lambda (cfr. capitolo 16). Si noti inoltre che non è possibile dichiarare la classe `Inner` con un modificatore come `private` o `public`. Trovandosi infatti all'interno di un metodo, tale dichiarazione non avrebbe senso.

6. Se viene dichiarata statica diventa automaticamente una **top-level**

class. Questo significa che non sarà più definibile come **classe interna** e non godrà della proprietà di poter accedere alle variabili d'istanza della classe in cui è definita. Potrà accedere solo ai membri statici dichiarati nella classe che la contiene. Infatti si definisce **classe interna**, una classe innestata che non è statica.

7. Solo se dichiarata statica può dichiarare membri statici.
8. Può essere dichiarata astratta.
9. Nei metodi di una classe interna è possibile utilizzare il reference `this`. Con esso ci si può riferire ai membri della classe interna e non della classe che la contiene. Per referenziare un membro della classe esterna bisognerebbe utilizzare la seguente sintassi:

```
NomeClasseEsterna.this.nomeMembroDaRefere
```

Questo è necessario solo nel caso ci sia possibilità d'ambiguità tra i membri della classe interna e i membri della classe esterna. Infatti, è possibile avere una classe interna ed una esterna che dichiarano un membro con lo stesso identificatore (supponiamo una variabile d'istanza `pippo`). All'interno dei metodi della classe interna, se non specifichiamo un reference per la variabile `pippo` verrà considerato implicitamente il reference `this`, quindi verrà referenziata la variabile d'istanza della classe interna. Per referenziare la variabile della classe esterna bisogna quindi utilizzare la sintassi:

```
NomeClasseEsterna.this.nomeMembroDaRefere
```

Per esempio, l'output del seguente codice:

```
public class Outer2 {
    private String stringa = "esterna";
    public class Inner2 {
        private String stringa = "interna";
        public void metodoInner() {
            System.out.println(Outer2.this.stringa + " " +
this.stringa);
        }
    }
```

```
     public static void main(String [] args) {
    Outer2 outer = new Outer2();
         Outer2.Inner2 inner = outer.new Inner2();
         inner.metodoInner();
     }
 }
```

sarà il seguente:

```
esterna interna
```

10. Infine, notiamo che se compiliamo una classe che contiene una classe interna, saranno creati due file: NomeClasseEsterna.class e NomeClasseEsterna$NomeClasseInterna.class.

Esiste anche un altro tipo di classe innestata detta **classe anonima**, e sarà argomento del paragrafo 10.2.

> **Le precedenti regole sono applicate anche ad altre strutture dati non ancora affrontate come le interfacce, le enumerazioni e le annotazioni. Non solo sarà possibile creare in una classe classi interne, interfacce interne, enumerazioni interne e annotazioni interne, ma lo si potrà fare anche in un'interfaccia, in un'enumerazione o in un'annotazione. Tuttavia stiamo parlando di pratiche di programmazione per ora trascurabili.**

10.1.3 Quando usare le classi innestate

In generale si può sempre fare a meno delle classi innestate dal punto di vista della progettazione. Il vantaggio consiste soprattutto nel risparmio di codice in determinate situazioni (come per esempio quando dobbiamo creare un gestore di eventi per un'interfaccia grafica come spiegato nell'appendice Q dedicata alle GUI). Potrebbe essere desiderabile creare una classe innestata solo in casi in cui ci sia una forte relazione esclusiva di interoperabilità tra due classi, tale da voler creare una dipendenza biunivoca tra le due. Per esempio una classe `Auto`, potrebbe avere bisogno di una classe `Meccanico`

che effettui delle riparazioni. Un meccanico dovrà avere accesso alla macchina per poterla aggiustare (dipendenza biunivoca). Supponendo di calarci in un ben determinato contesto dove un meccanico abbia sempre un'auto di riferimento, allora potrebbe apparire lecito dichiarare la classe `Meccanico` come classe interna. Infatti la soluzione senza classe innestata richiederebbe la codifica delle seguenti classi:

```
public class AutoNoInner {
    private String statoMotore;
    private MeccanicoNoInner meccanico;

    public AutoNoInner() {
        meccanico = new MeccanicoNoInner(this);
    }

    public void setStatoMotore(String statoMotore) {
        this.statoMotore = statoMotore;
    }
    public String getStatoMotore() {
        return statoMotore;
    }
}
public class MeccanicoNoInner {
    private AutoNoInner auto;

    public MeccanicoNoInner(AutoNoInner auto) {
        this.auto = auto;
    }

    public void aggiustaMotore() {
        auto.setStatoMotore("buono");
    }
}
```

Invece, creando `Meccanico` come classe interna, la situazione si semplifica molto:

```
public class Auto {
    private String statoMotore;

    public class Meccanico {
        public void aggiustaMotore() {
            statoMotore = "buono";
```

```
        }
    }
}
```

Il lato negativo di quest'approccio, è che se la classe `Meccanico` dovrà poi essere usata anche fuori dalla classe `Auto`, bisognerà sempre istanziare un'auto per istanziare un meccanico come spiegato nel paragrafo precedente, con la seguente sintassi:

```
Auto = new Auto();
Auto.Meccanico meccanico = auto.new Meccanico();
```

10.2 Classi anonime: definizione

Come le classi innestate, le **classi anonime** sono state introdotte successivamente alla nascita del linguaggio: nella versione 1.2. Le classi anonime non sono altro che classi innestate senza nome. Essendo classi innestate, godono delle stesse proprietà (cfr. paragrafo 10.1.2) e sono utilizzate per gli stessi scopi (soprattutto per la gestione degli eventi sulle interfacce grafiche). A differenza delle classi interne però, la dichiarazione di una classe anonima richiede anche che:

1. contestualmente alla dichiarazione della classe sia istanziato un suo oggetto;
2. l'esistenza di una sua superclasse o di una sua superinterfaccia di cui sfrutterà il costruttore (solo virtualmente nel caso di un'interfaccia). Se una classe non ha nome, non può avere un costruttore.

La sintassi a prima vista può disorientare:

```
public class Outer4 {
    private String messaggio = "Nella classe ";
    //Definizione della classe anonima e sua istanza
    ClasseEsistente ce = new ClasseEsistente() {
        @Override
        public void metodo() {
            System.out.println(messaggio + "anonima");
        }
    }; //si noti il ";"
    //...
```

```
}
//Superclasse della classe anonima
public class ClasseEsistente {
    public void metodo() {
        System.out.println("Nella classe esistente");
    }
}
```

Quando si dichiara una classe anonima la si deve anche istanziare, ma come istanziare una classe senza nome? Come si chiama il costruttore di una classe che non ha nome? Una classe anonima deve quindi estendere sicuramente un'altra classe (nell'esempio estendeva `ClasseEsistente`) e sfruttarne un reference (grazie al polimorfismo per dati) e un costruttore (per definizione la classe anonima non ha nome e quindi non ha costruttore). Infatti la classe anonima dell'esempio usa il reference `ce` di tipo `ClasseEsistente` per referenziare un oggetto della sua sottoclasse anonima, e implicitamente come tutte le sottoclassi chiama sempre il costruttore della superclasse.

Se compiliamo una classe che contiene una classe anonima, saranno creati due file: NomeClasseEsterna.class **e** NomeClasseEsterna$1.class. **E se introduciamo una seconda classe anonima, sarà creato anche il file** NomeClasseEsterna$2.class, **e così via per altre eventuali classi anonime.**

Si noti che una classe anonima viene sempre dichiarata con lo scopo di fare override di uno o più metodi della classe che estende. Se infatti nell'esempio avessimo definito un nuovo metodo (non ereditato) nella classe anonima, non saremmo stati in grado di invocarlo senza un reference del tipo della classe anonima (che però non può esistere). Per esempio, se nell'esempio precedente avessimo definito la classe anonima in questo modo:

```
//Definizione della classe anonima e sua istanza
ClasseEsistente ce = new ClasseEsistente() {
    @Override
    public void metodo() {
        System.out.println(messaggio + "anonima");
    }
```

```
    public void metodoNonInvocabile!() {
        System.out.println("Per una classe anonima non
esistono"
          + " reference: impossibile chiamare questo
metodo!");
    }
};
//...
```

per quanto detto nel capitolo 8 riguardante il polimorfismo, essendo la precedente una classe senza nome, non possiamo ottenere un reference della classe stessa, ma solo usarne uno della classe che estende.

Infine, come le classi interne, anche le classi anonime si possono dichiarare all'interno di metodi, ed anche in questo caso c'è il vincolo di poter accedere alle variabili locali e ai parametri solo se dichiarati `final`, o se sono effettivamente non modificabili (effectively final). Supponendo che sia stata definita la classe `Outer4`, il seguente codice è valido:

```
public class Outer4 {
    private String messaggio = "Nella classe ";
    public void metodoConClasse(final int a) {
        ClasseEsistente ce = new ClasseEsistente() {
            @Override
            public void metodo() {
                System.out.println(messaggio + "anonima
numero " + a);
            }
        };
        ce.metodo();
    }
}
```

In questo caso quindi, dichiarare `final` il parametro non è necessario affinché la classe anonima la possa utilizzare (questo vale da Java 8 in poi), ma di certo non è una cattiva abitudine. In generale comunque, dichiarare un parametro `final` anche se apparentemente inutile, potrebbe rinforzare l'idea che esso non si debba cambiare. Nel caso il parametro sia un reference inoltre, dichiarandolo `final` ci metteremmo al riparo già in fase di compilazione da eventuali problemi di assegnazioni inutili (che probabilmente porterebbero a un bug). Infatti come già visto in precedenza,

l’assegnazione ad un reference di un nuovo indirizzo non implica che il reference uscito dal metodo abbia davvero cambiato indirizzo (cfr. paragrafo 3.4.1 sul passaggio di parametri per valore). Ma è anche vero che quando abbiamo a che far con reference, anche se non possiamo cambiare il loro indirizzamento, è sempre possibile cambiare il loro stato interno tramite un metodo mutator (setter). Abbiamo affermato precedentemente che le regole per le classi innestate valgono anche per le interfacce, le enumerazioni e le annotazioni. Per quanto riguarda le classi anonime, è invece possibile solo estendere classi o interfacce, ma non enumerazioni o annotazioni. Se consideriamo per esempio l’interfaccia `Volante`, introdotta nel capitolo 6, aggiungendoci il nuovo metodo `plana()`:

```
public interface Volante {
    void plana();
    void decolla();
    void atterra();
}
```

allora sarà possibile crearne istanze non identificate in modo immediato, mediante la seguente sintassi:

```
public class TestVolanteAnonymous {
    public static void main(String args[]) {
        Volante ufo = new Volante() {
            @Override
            public void decolla() {
                System.out.println("Un oggetto non
identificato sta "
                                    + "decollando");
            }
            @Override
            public void plana() {
                System.out.println("Un oggetto non
identificato sta "
                                    + "planando");

            }
            @Override
            public void atterra() {
                System.out.println("Un oggetto non
identificato atterra...");
            }
```

```
        };
        //Usiamo l'oggetto della classe anonima.
        ufo.decolla();
        ufo.plana();
        ufo.atterra();
    }
}
```

Si noti come sembra di istanziare oggetti direttamente dall'interfaccia, che viene implementata contestualmente all'istanza dell'oggetto. Si noti anche che la sintassi sembra ci faccia utilizzare il costruttore di un'interfaccia! Ma sappiamo che le interfacce non hanno costruttori, quindi con la sintassi:

```
new Volante() { //...
```

stiamo usando un costruttore virtuale. È possibile anche estendere classi astratte per ridefinire i metodi astratti.

10.2.1 Quando usare le classi innestate

Un utilizzo particolarmente utile delle classi anonime è quando vengono dichiarate al momento di passare un parametro in un metodo. Probabilmente questo è il modo più utile per utilizzarle. Stiamo parlando di creare delle specializzazioni con override create al volo. Per esempio consideriamo la seguente classe `Pilota` che dichiara un metodo che prende in input un oggetto `Volante`:

```
public class Pilota {
    private String nome;

    public Pilota(String nome) {
        this.nome = nome;
    }

    public void fattiUnGiro(Volante volante) {
        volante.decolla();
        volante.plana();
        volante.atterra();
    }

    public void setNome(String nome) {
        this.nome = nome;
```

```
    }
    public String getNome() {
        return nome;
    }
}
```

Possiamo usare le classi anonime in maniera dinamica come nel seguente esempio:

```
public class TestPilota {
    public static void main(String args[]) {
       Pilota pilota = new Pilota("Simone");
       pilota.fattiUnGiro(new Volante() {
           @Override
           public void decolla() {
               System.out.println("Un oggetto non
identificato sta "
                                   + "decollando");
           }
           @Override
           public void plana() {
               System.out.println("Un oggetto non
identificato sta "
                                   + "planando");
           }
           @Override
           public void atterra() {
               System.out.println("Un oggetto non
identificato sta "
                                   + "atterrando");
           }
       } );
    }
}
```

In questo caso possiamo cambiare al volo l'implementazione dei metodi, senza aver magari mai dichiarato la classe, e senza neanche mai aver dato un reference all'oggetto creato. È un modo di lavorare meno object oriented ma molto utile. Possiamo dire di aver passato ad un metodo del codice da eseguire! Questo è davvero molto potente, cerchiamo di capire il perché.

Uno dei così detti "anti-pattern" della programmazione (il contrario dei pattern, ovvero qualcosa da evitare) è il cosiddetto "copia-incolla". Certo è comodo evitare di scrivere due volte lo stesso pezzo di codice, ma se lo

stiamo facendo, dobbiamo renderci conto che stiamo sbagliando a priori: dovremmo addirittura sentirci in colpa! Questo perché in effetti stiamo aprendo la strada a potenziali bachi. Infatti, se un giorno modificheremo uno dei due pezzi di codice perché ci accorgiamo di un bug, è probabile che non lo faremo nell'altro che rimarrà bacato. Dopotutto uno dei paradigmi fondamentali dell'Object Orientation e della buona programmazione in generale è il riuso. Il nostro scopo deve essere quindi quello di evitare a tutti i costi il copia-incolla.

Per esempio se ci troviamo a dover riscrivere un pezzo di codice in due classi diverse, è possibile che queste due classi possano trovarsi in qualche relazione di ereditarietà. Se avessimo la classe `Pilota` e `Persona` che dichiarano le proprietà (attributi privati con metodi accessor e mutator) `nome`, `cognome` e `codiceFiscale`, è facile pensare che `Pilota` possa estendere la classe `Persona`.

Invece, se due classi condividono in due metodi diversi un frammento di codice identico è molto probabile che quel pezzo di codice si possa astrarre in un metodo da invocare, magari spostato in una terza classe di utilità.

Se invece il pezzo di codice in comune è in due metodi della stessa classe, allora potrebbe essere possibile estrapolare questo pezzo di codice in un metodo privato che viene semplicemente invocato dai due metodi originari.

E se questo pezzo di codice in comune differisce di poco (magari di una sola riga) da un altro? In alcuni casi è comunque possibile astrarre in un metodo il frammento di codice, facendo dipendere il frammento di codice differente da un parametro passato al nuovo metodo. Cerchiamo di spiegarci meglio con un esempio. Per esempio consideriamo la seguente classe `Film`:

```
public class Film {
    private String nome;
    private String genere;
    private int mediaRecensioni;

    public Film(String nome, String genere, int
mediaRecensioni) {
        this.nome = nome;
        this.genere = genere;
        this.mediaRecensioni = mediaRecensioni;
    }
    // Metodi set e get omessi...
```

```
    public String toString(){
        return getNome();
    }
}
```

Andiamo a scrivere una classe `Videoteca`, che contiene due metodi praticamente uguali a parte una riga di codice:

```
public class Videoteca !!{
    private Film[] films;

    public Videoteca() {
        films = new Film[10];
        caricaFilms();
    }

    public void setFilms(Film[] films) {
        this.films = films;
    }
    public Film[] getFilms() {
        return films;
    }

    public Film[] getFilmDiFantascienza() {
        Film [] filmDiFantascienza = new Film[10];
        for (int i = 0, j= 0; i< 10;i++) {
            if ("Fantascienza".equals(films[i].getGenere()))
{
                filmDiFantascienza[j] = films[i];
                j++;
            }
        }
        return filmDiFantascienza;
    }

    public Film[] getBeiFilm() {
        Film [] beiFilms = new Film[10];
        for (int i = 0, j= 0; i< 10;i++) {
            if (films[i].getMediaRecensioni() > 3) {
                beiFilms[j] = films[i];
                j++;
            }
        }
        return beiFilms;
    }
```

```
    private void caricaFilms() {
        //Caricamento film...
    }
}
```

I metodi `getFilmDiFantascienza()` e `getBeiFilm()` che filtrano rispettivamente i film di fantascienza e i film con una media di recensione superiore a 3, sono praticamente identici a meno dei nomi delle variabili locali (che però sono solo nomi, e potremmo anche chiamare allo stesso modo) e il controllo nella clausola `if`. L'algoritmo non è nemmeno banale perché nel ciclo `for` abbiamo definito due indici che vengono incrementati in momenti diversi: `i` per scorrere l'array `films` (variabile d'istanza), mentre `j` l'array da riempire e ritornare. Sembra un ostacolo insormontabile evitare il copia-incolla, ed infatti nella maggior parte dei casi il programmatore Java medio lascia il codice così, mettendo a rischio di bug il programma.

Con la seguente classe stampiamo i film di fantascienza:

```
public class TestVideoteca {
    public static void main(String args[]) {
        Videoteca videoteca = new Videoteca();
        System.out.println("Film di Fantascienza:");
        Film[] filmDiFantascienza =
videoteca.getFilmDiFantascienza();
        for (Film film: filmDiFantascienza) {
            if (film != null) {
                System.out.println(film);
            }
        }
    }
}
```

La soluzione consiste nell'utilizzo di classi anonime. Per prima cosa dobbiamo creare una semplice interfaccia in questo modo che astrae il concetto di filtro:

```
public interface FiltroFilm {
    boolean filtra(Film film);
}
```

Poi eliminiamo i due metodi incriminati e li sostituiamo con il seguente:

```
public Film[] getFilmFiltrati(FiltroFilm filtroFilm) {
    Film [] filmFiltrati = new Film[10];
    for (int i = 0, j= 0; i< 10;i++) {
        if (filtroFilm.filtra(films[i])) {
            filmFiltrati[j] = films[i];
            j++;
        }
    }
    return filmFiltrati;
}
```

Si noti che nel codice precedente abbiamo evidenziato l'invocazione di un metodo virtuale. Infatti il metodo `filtra()` è astratto e invocato su un reference di un'interfaccia. Quindi sfruttando quanto abbiamo imparato nel capitolo 8, per filtrare i nostri film possiamo utilizzare il seguente codice:

```
//...
Videoteca videoteca = new Videoteca();
System.out.println("Bei Film:");
Film[] beiFilms = videoteca.getFilmFiltrati(new FiltroFilm()
{
    @Override
    public boolean filtra(Film film) {
        return film.getMediaRecensioni() > 3;
    }
});
//...
System.out.println("\nFilm di Fantascienza:");
Film[] filmDiFantascienza = videoteca.getFilmFiltrati(new
FiltroFilm() {
    @Override
    public boolean filtra(Film film) {
        return "Fantascienza".equals(film.getGenere());
    }
});
//...
```

In questo modo abbiamo passato al volo il codice da eseguire al metodo come parametro! E abbiamo evitato il copia-incolla. Possiamo anche inventare nuovi filtri definendoli al volo.

Questa soluzione è molto avanzata e flessibile, forse è la più complessa che abbiamo visto sino ad ora, tanto che molti programmatori Java non l'hanno mai usata! Questo tipo di soluzione è anche molto usata per la

gestione delle azioni sulle interfacce grafiche come è possibile leggere sull'appendice on line dedicata.

Il lato negativo di quest'approccio alla programmazione è che non è favorita la leggibilità e bisogna scrivere molto codice. Ma con l'introduzione delle espressioni lambda in Java 8, vedremo nel capitolo 16 che sarà possibile ottenere lo stesso risultato con un minore sforzo e una sintassi molto più compatta e potente.

10.3 Tipi Enumerazioni

Gli **enumerated types**, che traduciamo come **tipi enumerazioni**, o più semplicemente solo con **enumerazioni** o **enum**, sono uno dei componenti fondamentali della programmazione Java che non abbiamo ancora introdotto. Si tratta di una nuova tipologia di struttura dati, che si aggiunge alle classi, alle interfacce e alle annotazioni (che approfondiremo in un capitolo dedicato), e quindi si possono salvare in file con suffisso .java. Questa volta invece di utilizzare `class` o `interface`, useremo la parola chiave: **enum.** Le enumerazioni sono strutture dati somiglianti alle classi, ma con proprietà particolari. Facciamo subito un esempio:

```
public enum MiaEnumerazione {
    UNO, DUE, TRE;
}
```

Abbiamo appena definito un'enum di nome `MiaEnumerazione`, definendo tre suoi **elementi** (detti anche **valori)** che si chiamano `UNO, DUE, TRE.` È come se avessimo definito un nuovo tipo. Infatti, un'enumerazione viene trasformata dal compilatore in una classe che estende la classe astratta `Enum` (package `java.lang`). Gli elementi di questa enum sono implicitamente di tipo `MiaEnumerazione` e quindi non va specificato il tipo. Essi sono semplicemente definiti separandoli con virgole. Si tratta di costanti statiche, ma non bisogna dichiararle né `final` né `public` né `static` in quanto in un'enum lo sono già implicitamente. Si noti che nell'esempio dopo la lista dei valori il ";" non è necessario, ma è consigliato. Diventerà infatti necessario nel caso siano definiti altri elementi nell'enumerazione, come per esempio metodi.

Ogni elemento di `MiaEnumerazione` è di tipo `MiaEnumerazione.` È questa

la caratteristica delle enum più difficile da digerire all'inizio. Quindi, definita un'enumerazione, si definiscono anche tutte le sue possibili istanze. Non si possono istanziare altre `MiaEnumerazione`, oltre a quelle definite da `MiaEnumerazione` stessa. Ecco perché sembra essere più corretto definire i suoi elementi come "valori" dell'enumerazione. Trattandosi di costanti, la convenzione Java (cfr. capitolo 3) consiglia di definire gli elementi di un'enumerazione con caratteri maiuscoli.

Ricordiamo che come separatore di parole può essere utilizzato il carattere underscore ("_"). Inoltre le enumerazioni vanno definite con la stessa convenzione delle classi. La sintassi di una enum, purtroppo, non si limita solo a quanto appena visto. Esistono tante altre caratteristiche che può avere un'enumerazione, come implementare interfacce, definire costruttori, metodi, etc.

10.3.1 Ereditarietà e polimorfismo con `enum`

Una enum non si può estendere, né può estendere un'altra enum o un'altra classe. Infatti le enumerazioni verranno trasformate dal compilatore in classi che estendono la classe **java.lang.Enum.** Questo significa che erediteremo dalla classe `java.lang.Enum` diversi metodi, che possiamo tranquillamente invocare sugli elementi della enum. Per esempio il metodo `name()` è definito in modo tale da restituire il nome dell'elemento, per cui:

```
System.out.println(MiaEnumerazione.UNO.name());
```

produrrà il seguente output:

```
UNO
```

Anche il metodo `toString()` restituisce lo stesso risultato, quindi l'istruzione:

```
System.out.println(MiaEnumerazione.UNO);
```

produce esattamente lo stesso output (visto che il metodo `println()` chiama il metodo `toString()` sull'oggetto in input). È anche possibile fare override dei metodi di `java.lang.Enum` (vedremo tra poco come), compreso il metodo `toString()`. Invece, del metodo `name()` non si può fare override, perché è

dichiarato `final`. `Enum` dichiara anche un metodo complementare a `toString()`: il metodo statico `valueOf()`, che, passandogli in input una stringa, ritorna il corrispondente valore dell'enumerazione. Per esempio:

```
MiaEnumerazione elemento = MiaEnumerazione.valueOf("UNO");
System.out.println(elemento);
```

stamperà:

```
UNO
```

Inoltre `Enum` definisce il metodo `final ordinal()`. Tale metodo ritorna la posizione all'interno della enum di un suo elemento. Come al solito l'indice parte da zero e questo metodo è sfruttabile all'interno di cicli.

Le enumerazioni definiscono anche il metodo `values()`. Questo metodo consente di iterare sui valori di un'enumerazione. Tale pratica potrebbe servire quando non conosciamo l'enumerazione in questione. Per esempio, sfruttando un ciclo for migliorato, possiamo stampare i contenuti dell'enumerazione `MiaEnumerazione`:

```
for (MiaEnumerazione miaEnumerazione :
MiaEnumerazione.values()) {
    System.out.println(miaEnumerazione + " è in posizione "
      + miaEnumerazione.ordinal());
}
```

L'output del precedente esempio è:

```
UNO è in posizione 0
DUE è in posizione 1
TRE è in posizione 2
```

Come abbiamo già asserito, le enumerazioni sono trasformate dal compilatore in classi che estendono la classe `java.lang.Enum`. In particolare, ogni enum estende implicitamente la classe astratta `java.lang.Enum` (che non è un'enumerazione). Ecco perché possono usufruire o sottoporre ad override i metodi di `Enum`. Attenzione che il compilatore non permetterà allo sviluppatore di creare classi che estendono direttamente la classe `Enum`. Essa è una classe speciale creata appositamente per supportare il concetto di enumerazione.

Un’enumerazione non può estendere altre enumerazioni né altre classi. Infatti, se dovesse estendere un’altra enumerazione o un’altra classe, il compilatore non potrebbe fare estendere ad essa la classe `java.lang.Enum`, per le regole dell’ereditarietà singola che si applica alle classi.

È invece possibile far implementare una o più interfacce ad una enum. Infatti, possiamo dichiarare in una enum tutti i metodi che vogliamo, compresi quelli da implementare di un’interfaccia. Questo significa che se vogliamo sfruttare il polimorfismo per dati, possiamo farlo solo tramite l’utilizzo di interfacce o un reference di tipo enum. Per esempio, consideriamo la seguente interfaccia:

```
public interface Numeratore {
    void stampaIndice();
}
```

riscriviamo l’enumerazione facendogli implementare l’interfaccia precedente:

```
public enum MiaEnumerazione2 implements Numeratore {
    UNO, DUE, TRE;
    @Override
    public void stampaIndice() {
        System.out.println("Indice: " + this.ordinal());
    }
}
```

e sarà possibile usare il polimorfismo per dati sfruttando l’interfaccia `Numeratore` nel seguente modo:

```
Numeratore n = MiaEnumerazione2.DUE;
n.stampaIndice();
```

che darà come output:

```
Indice: 1
```

È possibile anche usare un reference di tipo `java.lang.Enum` per utilizzare il polimorfismo per dati:

```
Enum e = MiaEnumerazione2.UNO;
```

Ma con un reference di tipo `java.lang.Enum`, non potremo chiamare il metodo `stampaIndice()` come visto precedentemente, per le regole del

polimorfismo per dati.

> **L'ultima istruzione sarà sì compilata, ma sarà segnalato un warning dal compilatore. Infatti non è stata utilizzata correttamente la classe `Enum` che è definita usando un "tipo generico". Spiegheremo dettagliatamente questo argomento più avanti nel prossimo capitolo.**

Non potendo estendere una enum, non la si potrà dichiarare `abstract`. Quindi, nel momento in cui implementiamo un'interfaccia in una enum, dovremo implementare obbligatoriamente tutti i metodi ereditati.

10.3.2 Metodi, variabili, costruttori e tipi innestati in una enumerazione

In una enum è possibile anche creare variabili, metodi e costruttori. Questi ultimi però sono implicitamente dichiarati `private` e non è possibile utilizzarli se non nell'ambito dell'enumerazione stessa. Per esempio, con il seguente codice viene ridefinita l'enum `MiaEnumerazione` per la terza volta:

```
public enum MiaEnumerazione3 {
    ZERO(), UNO(1), DUE(2), TRE(3);
    private int valore;
    private MiaEnumerazione3() {
    }
    MiaEnumerazione3(int valore) {
        setValore(valore);
    }
    public void setValore(int valore) {
        this.valore = valore;
    }
    public int getValore() {
         return this.valore;
    }
    @Override
    public String toString() {
        return "" + valore;
    }
```

```
}
```

Si noti come sia sfruttato il costruttore quando vengono definiti gli elementi dell'enum.

Per le enumerazioni valgono le seguenti regole.

1. **Qualsiasi dichiarazione deve seguire la dichiarazione degli elementi dell'enumerazione.** Se anteponessimo una qualsiasi dichiarazione alla lista degli elementi, otterremo un errore in compilazione. In questo caso la dichiarazione degli elementi deve terminare esplicitamente con un "`;`", ed è buona abitudine che sia sempre così.
2. **È possibile dichiarare un qualsiasi numero di costruttori (sfruttando l'overload), che implicitamente saranno considerati** `private`. Nell'esempio abbiamo due costruttori, di cui per uno abbiamo esplicitato (ma è ridondante) il modificatore `private`. Se avessimo dichiarato esplicitamente un costruttore `public` o `protected`, avremmo ottenuto un errore in compilazione. Per il resto valgono le regole applicate ai costruttori delle classi. Come per le classi anche per le enumerazioni, se non inseriamo costruttori, il compilatore ne aggiungerà uno per noi senza parametri (il costruttore di default). Sempre come per le classi, il costruttore di default non verrà inserito nel momento in cui ne inseriamo noi uno esplicitamente, come nell'esempio precedente.
3. **Quando sono esplicitati i costruttori come in questo caso, i valori dell'enum possono utilizzarli.** Basta osservare il codice dell'esempio. I valori `UNO`, `DUE` e `TRE` utilizzano il costruttore che prende in input un intero (la sintassi è intuitiva). Il valore `ZERO`, invece, utilizza il costruttore senza parametri. In particolare è possibile notare come il valore `ZERO` sia dichiarato in maniera diversa rispetto agli altri valori. Infatti, esplicita due parentesi vuote, che sottolineano come stia utilizzando il costruttore senza parametri. Tale sintassi è assolutamente superflua (le parentesi tonde potevano essere omesse) ed è stata riportata solo per preparare il lettore a strane sorprese. Se nell'esempio avessimo avuto un unico costruttore (quello

che prende come parametro in input un intero) tutti gli elementi dell'enum avrebbero obbligatoriamente dovuto utilizzare quell'unico costruttore.

4. **Nelle enumerazioni è possibile dichiarare tipi innestati (classi, interfacce, enumerazioni e annotazioni) esattamente come lo si può fare nelle classi, nelle interfacce (e nelle annotazioni).** Non bisogna fare alcuna attenzione particolare a queste dichiarazioni, se non come già asserito, nel posizionarle dopo i valori dell'enum. Per esempio, è possibile anche dichiarare un'enum in un'enum, come segue:

```
public enum MyEnum {
    ENUM1 (), ENUM2;
    public enum MyEnum2 {a, b, c}
}
```

Non è possibile invece fare il contrario: in una classe innestata non è ammesso dichiarare un'enumerazione.

10.3.3 Quando utilizzare una `enum`

Non si può dire che le enum permettano di creare codice che non si può creare con le classi, semmai si può affermare che consentiranno di creare codice più robusto in alcuni casi. È consigliato l'uso dell'enum ogni qualvolta ci sia bisogno di dichiarare un numero finito di valori da utilizzare per gestire il flusso di un'applicazione. Facciamo un esempio. Prima dell'avvento delle enumerazioni in Java, spesso si rendeva necessario creare costanti simboliche per rappresentare valori. Tali costanti venivano spesso definite di tipo intero oppure stringa. A volte si raccoglievano le costanti all'interno di interfacce dedicate. Per esempio:

```
public interface Azione {
    public static final int AVANTI = 0;
    public static final int INDIETRO = 1;
    public static final int FERMO = 2;
}
```

Oltre al fatto che le interfacce si potrebbero usare per scopi più object oriented, il motivo fondamentale per cui questo approccio è considerato sconsigliabile, è che a livello di compilazione non si possono esplicitare vincoli che impediscano all'utente di utilizzare scorrettamente un tipo. Consideriamo per esempio il seguente codice:

```
public void esegui(int azione) {
    switch (azione) {
        case Azione.AVANTI:
            vaiAvanti();
        break;
        case Azione.INDIETRO:
            vaiIndietro();
        break;
        case Azione.FERMO:
            fermati();
        break;
   }
}
```

Nessun compilatore potrebbe rilevare che la seguente istruzione non è valida:

```
oggetto.esegui(3);
```

È possibile aggiungere una clausola `default` al costrutto `switch`, dove si gestisce in qualche modo il problema, ma questo sarà eseguito solo in fase di runtime dell'applicazione, e non in fase di compilazione. Un'implementazione più robusta della precedente potrebbe richiedere l'utilizzo di una classe come la seguente:

```
public class Azione {
    private String nome;
    public static final Azione AVANTI = new
Azione("AVANTI");
    public static final Azione INDIETRO = new
Azione("INDIETRO");
    public static final Azione FERMO = new Azione("FERMO");
    public Azione(String nome) {
         setNome(nome);
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
```

```
    public String getNome() {
        return this.nome;
    }
}
```

In tale caso il metodo `esegui()` potrebbe essere modificato in modo tale da sostituire il costrutto `switch` con un costrutto `if`:

```
public void esegui(Azione azione) {
    if (azione == Azione.AVANTI) {
        vaiAvanti();
    }
    else if (azione == Azione.INDIETRO) {
        vaiIndietro();
    }
    else if (azione == Azione.FERMO) {
        fermo();
    }
}
```

Purtroppo anche in questo caso però il problema rimane (anche se l'astrazione con una classe sembra migliore rispetto a quella con un'interfaccia). È sempre possibile passare `null` al metodo `esegui()`. Qualsiasi controllo volessimo inserire, potrebbe risolvere il problema solo in fase di runtime dell'applicazione. In casi come questo l'uso di un'enumerazione mette tutti d'accordo. Se infatti creiamo la seguente enumerazione:

```
public enum AzioneEnum {
    AVANTI, INDIETRO, FERMO;
}
```

senza modificare il codice del metodo `esegui()` (è possibile utilizzare un tipo enum anche come variabile di test di un costrutto `switch`) avremo risolto il problema.

10.3.4 Enumerazioni innestate (in classi) o enumerazioni membro

Come già asserito, anche le enumerazioni si possono innestare nelle classi. Per esempio, è possibile scrivere codice come il seguente:

```
public class Volume {
    public enum Livello {ALTO, MEDIO, BASSO};
    // implementazione della classe...
}
```

Se volessimo stampare un elemento della enumerazione all'interno della classe con il metodo `toString()`, è possibile utilizzare la seguente sintassi:

```
System.out.println(Livello.ALTO);
```

Nel caso in cui ci si trovasse al di fuori della classe, dovremmo utilizzare la sintassi:

```
System.out.println(Volume.Livello.ALTO);
```

È anche possibile utilizzare l'incapsulamento, ma trattandosi di costanti statiche non si corrono grossi pericoli, dato che il loro valore non può cambiare. Una enum innestata è statica implicitamente. Infatti, il seguente codice è valido:

```
public class Volume {
    public enum Livello {ALTO, MEDIO, BASSO};
    // implementazione della classe...
    public static void main(String args[]) {
        System.out.println(Livello.ALTO);
    }
}
```

Se `Livello` non fosse statica non avremmo potuto utilizzarla direttamente in un metodo statico come il `main()`.

> **Su altri testi le enumerazioni innestate sono chiamate semplicemente "enumerazioni membro" (membro di una classe; cfr. capitolo 2). Per non avviare una sterile discussione su come sia più corretto chiamare tale costrutto, possiamo affermare che sono solo punti di vista differenti.**

Si noti come gli import statici descritti nel paragrafo 6.8.4, si adattino benissimo ad essere utilizzati per importare i valori delle enumerazioni, o

anche i tipi innestati all'interno delle enumerazioni. Infatti, entrambi sono elementi statici. Supponendo di dichiarare la classe `Volume` appartenente al package `musica`:

```
package musica;

public class Volume {
    //...
```

Per esempio in un'altra classe potremmo importare il valore `ALTO` dell'enum `Livello` innestata all'interno della classe `Volume`, con la seguente sintassi:

```
import static musica.Volume.Livello.ALTO;

public class TestStaticImport {
    public static void main(String args[]) {
        System.out.println(ALTO);
    }
}
```

Si noti che sarà stampato il risultato del metodo `toString()` chiamato sull'elemento `ALTO`.

> **In effetti gli import statici furono introdotti contestualmente alle enumerazioni nella versione 5 del linguaggio. Sembra proprio che gli import statici abbiano avuto soprattutto il compito di supportare l'introduzione delle enumerazioni, cercando di ridurne la verbosità nell'utilizzo.**

10.3.5 Enumerazioni e metodi specifici degli elementi

Ma è proprio impossibile estendere un'enum? In effetti no! È possibile infatti che una certa enumerazione sia estesa dai suoi stessi elementi sfruttando la sintassi delle classi anonime. Si possono definire metodi nell'enumerazione e fare override di essi con i suoi elementi. Consideriamo la seguente enumerazione `PuntiCardinali`:

```
public enum PuntiCardinali {
    NORD {
```

```
        @Override
        public void test() {
            System.out.println("metodo di NORD");
        }
    },
    SUD, OVEST, EST;
    public void test() {
        System.out.println("metodo dell'enum");
    }
}
```

È stato definito un metodo che abbiamo chiamato `test` () e dovrebbe stampare la stringa “metodo dell’enum”. Però l’elemento `NORD`, con una sintassi simile a quella delle classi anonime, dichiara anch’esso lo stesso metodo, sottoponendolo a override. Infatti, il compilatore tramuterà `NORD` proprio in una classe anonima, che estenderà `PuntiCardinali`. Quindi l’istruzione:

```
PuntiCardinali.NORD.test();
```

stamperà:

```
metodo di NORD
```

mentre l’istruzione:

```
PuntiCardinali.SUD.test();
```

stamperà:

```
metodo dell'enum
```

perché `SUD` non ha fatto override di `test()`.

10.3.6 `Switch` e `enum`

Nel capitolo 4 abbiamo introdotto il costrutto `switch`. Ora che conosciamo le enumerazioni dobbiamo capire come è possibile utilizzare questo concetto nel costrutto. Se un costrutto `switch` definisce come variabile di test un’enumerazione, tutte le istanze di tale enumerazione, possono essere possibili costanti per i case. Per esempio, tenendo presente l’enumerazione `Livello` di cui sopra, consideriamo il seguente frammento di codice:

```
switch (getLivello()) {
    case ALTO:
        System.out.println(Livello.ALTO);
    break;
    case MEDIO:
        System.out.println(Livello.MEDIO);
    break;
    case BASSO:
        System.out.println(Livello.BASSO);
    break;
}
```

La variabile di test è di tipo `Livello` e le costanti dei case sono gli elementi dell'enumerazione stessa. Si noti come gli elementi non abbiano bisogno di essere referenziati con il nome dell'enumerazione nel seguente modo:

```
case Livello.ALTO:
    System.out.println(Livello.ALTO);
    break;
case Livello.MEDIO:
    System.out.println(Livello.MEDIO);
    break;
case Livello.BASSO:
    System.out.println(Livello.BASSO);
    break;
```

infatti la variabile di test fornisce già la sicurezza del tipo. Nonostante in un costrutto `switch` che si basa su un'enum sia escluso che la clausola `default` possa essere eseguita durante il runtime (questo è uno dei vantaggi delle enum rispetto ai "vecchi" approcci) è comunque buona norma utilizzarne una. Infatti, è facile che l'enumerazione subisca nel tempo delle aggiunte. Questo è particolarmente vero se il codice è condiviso tra più programmatori. In tal caso ci sono due approcci da consigliare. Il primo è più soft e consiste nel gestire comunque eventuali nuovi tipi in maniera generica. Per esempio:

```
default:
    System.out.println(getLivello());
```

Il secondo metodo è senz'altro più robusto, in quanto basato sulle asserzioni:

```
default:
    assert false: "valore dell'enumerazione nuovo: " +
getLivello();
```

Si noti che l'asserzione dovrebbe fare emergere il problema durante i test se l'enumerazione `Livello` è stata ampliata con nuovi valori. Nel caso non si voglia implementare la clausola `default`, possiamo comunque farci aiutare dal compilatore ove l'enumerazione coinvolta nello `switch` si evolva. Infatti, se in uno `switch` non sono contemplati tutti i case dell'enumerazione, come per esempio:

```
case ALTO:
    System.out.println(Livello.ALTO);
break;
case BASSO:
    System.out.println(Livello.BASSO);
break;
```

allora in compilazione il compilatore ci segnalerà un warning (se stiamo utilizzando un IDE, oppure se stiamo usando l'opzione -Xlint per compilare da riga di comando (cfr. paragrafo 9.5.1)).

Riepilogo

Una **classe innestata** è una classe definita all'interno di un'altra classe. Le classi innestate possono accedere agevolmente ai membri (anche quelli statici) della classe che le contiene. Non possono però dichiarare membri statici. Per le classi innestate si possono usare gli specificatori d'accesso con le stesse regole utilizzate per i membri di una classe. Se una classe innestata non è dichiarata statica, viene detta anche **classe interna.** Le classi innestate statiche invece diventano **top-level class**, e non godranno della proprietà di poter accedere alle variabili d'istanza della classe in cui è definita, ma possono dichiarare membri statici, e accedere ai membri statici della classe in cui è dichiarata. Se si vuole istanziare una classe innestata (supponiamo si chiami `Inner`) dall'esterno della classe che la contiene (supponiamo si chiami `Outer`), bisogna farlo sfruttando una sintassi speciale. Dapprima si istanzia un oggetto della classe esterna (supponiamo tale oggetto si chiami `oggettoClasseEsterna`), poi bisogna istanziare la classe interna nel modo seguente: `Outer.Inner inner = outer.new Inner();`. Utilizzare il reference `this` all'interno di una classe innestata, implica referenziare l'oggetto corrente della classe innestata. Se si vuole utilizzare un reference

per puntare all'istanza dell'oggetto istanziato dalla classe esterna, bisogna utilizzare la sintassi: `NomeClasseEsterna.this`. Una classe innestata si può dichiarare anche all'interno di un metodo (o in generale in un blocco di codice) e in questo caso viene detta **classe locale.** In questo caso le variabili locali saranno accessibili solo se dichiarate `final` oppure **effettivamente non modificate.** Le classi innestate possono essere dichiarate astratte.

Una **classe anonima** è una classe innestata senza nome. Gode di tutte le proprietà di una classe interna, inoltre deve essere istanziata nel momento in cui si dichiara e deve essere sottoclasse di una classe esistente di cui sfrutterà il costruttore. Può essere usata per passare codice dinamicamente ad un metodo.

Le **enumerazioni** sono particolari strutture dati che definiscono anche tutti i propri **elementi** (un numero finito di elementi). Tali elementi sono implicitamente dichiarati `public`, `final` e `static`., così come i costruttori sono implicitamente dichiarati privati. Le enumerazioni vengono trasformate dal compilatore in classi che estendono la classe **java.lang.Enum**, di cui ereditano i metodi. Per tale ragione non possono estendere altre classi ma possono eventualmente implementare interfacce per sfruttare il polimorfismo. Possono contenere altre **enumerazioni innestate**, e spesso sono esse stesse enumerazioni innestate in classi. È possibile dichiarare metodi e ogni elemento può eventualmente fare override del metodo dichiarato all'interno dell'enumerazione, creando così **metodi specifici degli elementi dell'enumerazione.** Il costrutto `switch` supporta il test con enumerazioni, e l'istruzione `import static` può risultare utile per limitare la verbosità del codice che fa uso di enumerazioni.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data

Comprendere cosa sono i tipi innestati e i vantaggi che comportano (unità 10.1)	☐	
Saper elencare le proprietà fondamentali delle classi innestate (unità 10.1)	☐	
Saper definire ed utilizzare le classi anonime (unità 10.2)	☐	
Comprendere l'utilizzo e l'utilità delle enumerazioni (unità 10.3)	☐	
Comprendere l'utilizzo e l'utilità delle caratteristiche avanzate delle enumerazioni (unità 10.3)	☐	

Note:

11

Generics

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:
✔ Comprendere e saper utilizzare i tipi generici (unità 11.1).
✔ Comprendere la deduzione automatica del tipo, i raw type e gestire i warning del compilatore (unità 11.1).
✔ Comprendere l'erasure e come si applica l'ereditarietà con i tipi generici (unità 11.2).
✔ Saper usare le wildcard e i parametri bounded (unità 11.2, 11.3, 11.4).
✔ Saper creare i propri tipi generici (unità 11.3).
✔ Saper creare metodi generici (unità 11.3).
✔ Saper gestire le wildcard capture con i metodi helper (unità 11.3).
✔ Saper creare parametri covarianti (unità 11.4).

È normale che il codice abbia dei bug. Tuttavia esistono bachi che è possibile risolvere intercettandoli durante la compilazione. Questo si può fare se il linguaggio supporta strumenti come i **generics.** In particolare oltre a permettere al compilatore di individuare possibili bachi in fase di compilazione, i generics consentono di scrivere codice meno verboso

evitando pericolosi cast di tipi, e permettendo di creare algoritmi generici (ovvero basati sui generics). Si tratta in qualche modo quindi di introdurre un nuovo tipo di programmazione: la **programmazione generica.** Quindi i **tipi generici** (in inglese **generic types)** quando furono introdotti nella versione 5 del linguaggio, sconvolsero il modo di scrivere il codice Java. Con i **generics** la programmazione Java divenne più potente e robusta, ed oggi la libreria standard è basata su tantissime classi, interfacce e metodi generici. L'utilizzo più massiccio dei tipi generici è sicuramente contestuale all'utilizzo delle collection. In questo capitolo quindi introdurremo brevemente qualche collection che ci servirà come esempio, ma poi approfondiremo il loro studio nel capitolo 12 dedicato alla libreria `java.util`. L'utilizzo dei generics però, non si limita assolutamente alle collection.

11.1 Generics e tipi parametro

Quando dichiariamo una classe (o un'interfaccia) possiamo dichiararla come tipo generico (**generic type**) aggiungendo alla definizione uno o più **tipi parametro** (in inglese **parameter type).** La sintassi fa uso di parentesi angolari che circondano gli identificatori dei tipi:

```
class IdentificatoreDellaClasse <T1, T2,... Tn> {
    //...
}
```

Per esempio, consideriamo la seguente classe contenitore che contiene un oggetto qualsiasi:

```
public class Contenitore {
    private Object object;

    public void setObject(Object object) {
        this.object = object;
    }

    public Object getObject() {
        return object;
    }
}
```

Questa classe sembra semplice e ben fatta, ma non sarà così semplice da

gestire. Infatti, una volta recuperato l'oggetto `object` mediante il metodo `getObject()`, saremo obbligati a convertirlo per poterlo usare. Questa operazione è però potenzialmente pericolosa, perché potrebbe anche recuperare un tipo diverso da quello che ci aspettiamo. Per esempio, in un programma potremmo scrivere parti di codice che utilizzano questa classe in maniera incompatibile. In una parte del codice potremmo inserire all'interno di un oggetto di tipo `Contenitore` un oggetto di tipo `Integer` mediante il metodo `setObject()`, e poi da qualche altra parte tentare di recuperare un'oggetto `String` tramite il metodo `getObject()`. In casi come questo la Java Virtual Machine lancerà una `ClassCastException`. In generale per usare questa classe dovremmo scrivere istruzioni come le seguenti:

```
Contenitore contenitore = new Contenitore();
contenitore.setObject("Stringa");
// contenitore.setObject(new Integer(l));
String object = (String)contenitore.getObject();
System.out.println(object);
```

Si noti che è stato necessario il cast a stringa al momento del recupero dell'oggetto. Se avessimo impostato un oggetto `Integer` nel contenitore (commentando la seconda riga e decommentando la terza riga) avremmo ottenuto una `ClassCastException`. Per evitare l'eccezione avremmo potuto utilizzare l'operatore `instanceof`:

```
Object object = contenitore.getObject();
if (object instanceof String) {
    String string = (String)object;
    //...
} else if (object instanceof Integer) {
    Integer integer = (String)object;
    //...
} else if //...
```

ma il codice sarebbe diventato più verboso e incline all'errore. Per rendere il nostro codice più robusto dobbiamo rendere la classe generica nel seguente modo:

```
public class ContenitoreGenerics<T> {
    private T object;
```

```
    public void setObject(T object) {
        this.object = object;
    }

    public T getObject() {
        return object;
    }
}
```

Si noti che al posto di `Object` ora c'è il **tipo parametro** `T`, che non rappresenta un tipo esistente. Questo verrà sostituito con un tipo reale nel momento in cui verrà istanziato un `ContenitoreGenerics`. Per esempio:

```
ContenitoreGenerics<String> contenitore1 =
  new ContenitoreGenerics<String>();
contenitore1.setObject("Stringa"); //si possono passare solo
stringhe
String object = contenitore1.getObject(); //niente cast
System.out.println(object);
```

In questo modo è come se avessimo sostituito al parametro `T` la classe `String` in tutta la definizione della classe. Quindi il metodo `setObject()` accetterà solo stringhe; se provassimo a passare altri tipi otterremmo un errore in compilazione. Inoltre non è necessario un cast a `String` per recuperare l'oggetto tramite il metodo `getObject()`. In definitiva, per l'oggetto `contenitore!`, è come se la classe `ContenitoreGenerics` fosse stata dichiarata così:

```
public class ContenitoreGenerics<String> {
    private String object;

    public void setObject(String object) {
        this.object = object;
    }

    public String getObject() {
        return object;
    }
}
```

Si è solo sostituito il tipo parametro `T` con il tipo `String`.

La potenza della classe generica `ContenitoreGenerics` è che può essere

parametrizzata anche con qualsiasi altro tipo. Per esempio:

```
ContenitoreGenerics<Integer> contenitore2 =
  new ContenitoreGenerics<Integer>();
Contenitore2.setObject(new Integer("1"));
String object = contenitore2.getObject(); //niente cast
System.out.println(object);
```

Strano a dirsi, ma un tipo generico per certi versi è da considerare un tipo più specifico!

> **Per convenzione quando si dichiara un tipo parametro si usa un identificatore costituito da una sola lettera maiuscola, che dovrebbe rappresentare l'iniziale di un nome simbolico (nel caso di `T` significa "Type"). In particolare la libreria standard usa spesso: `E` per "Element", `K` per "Key", `N` per "Number", `V` per "Value", `T` per "Type", `S, U, V` per il secondo, terzo e quarto tipo.**

11.1.1 Generics e collection

L'utilizzo più classico che si fa dei tipi generici, è con le classi e le interfacce del **framework Collections.** Con la parola **framework**, intendiamo una micro architettura che mette a disposizione tipi estendibili nell'ambito di uno specifico dominio. Nella teoria dell'architettura software un framework è un esempio di **partizionamento orizzontale**, in altre parole una parte del software riutilizzabile ed estendibile. Ecco per esempio come è dichiarata l'interfaccia `List`:

```
public interface List<E> extends Collection<E>
```

Come già asserito nel paragrafo precedente, il tipo generico `E` definito per gli oggetti `List`, non è un tipo esistente, ma solo un identificatore generico sostituibile con qualsiasi altro tipo.

> **Si noti come l'interfaccia `List` estenda `Collection` dichiarata anch'essa generica.**

Siccome l'interfaccia `List` è un tipo generico, quando la si utilizza è possibile parametrizzarla con il tipo parametro che si vuole. Per esempio potremmo istanziare la sua più famosa implementazione, la sottoclasse `ArrayList` (introdotta nella nota conclusiva del paragrafo 8.3.2 riguardante le collezioni eterogenee), con la seguente sintassi:

```
List<Auto> lista = new ArrayList<Auto>();
```

Questo significa che potremo aggiungere a questa collezione solo oggetti di tipo `Auto`. Infatti il metodo `add()` che serve per aggiungere elementi alla lista è dichiarato così:

```
public boolean add(E o)
```

In questo modo una volta istanziato l'oggetto `lista` parametrizzandolo con il tipo `Auto`, il tipo parametro `E` diventa di tipo `Auto`, e quindi anche il parametro del metodo `add()` diventa di tipo `Auto`. Ovvero, per l'oggetto `lista`, è come se il metodo `add()` fosse dichiarato così:

```
public boolean add(Auto o)
```

Anche se una classe è definita generica, l'utilizzo dei tipi parametri non è obbligatorio. Il compilatore di un IDE però, segnalerà un warning in caso di non utilizzo (come vedremo in dettaglio nel paragrafo 11.1.6). Questo per permettere la compatibilità all'indietro con il codice scritto prima dell'avvento di Java 5, dove i generics semplicemente non esistevano.

Supponiamo quindi di aver creato un `ArrayList` in cui vogliamo inserire solo stringhe. Saremo obbligati ad utilizzare un casting quando si estrarranno le stringhe dall'`ArrayList` anche se si conosce a priori il tipo. Per esempio, consideriamo il seguente codice:

```
ArrayList list = getListOfStrings();
int size = list.size();
for (int i = 0; i < size; i++) {
    String stringa = (String)list.get(i);
}
```

Se rimuovessimo il cast, otterremmo un errore in compilazione come il seguente:

```
incompatible types
found :  java.lang.Object
required: java.lang.String
String stringa = list.get(i);
^
```

Infatti il compilatore non può sapere a priori che tipo di oggetti possono essere stati messi negli elementi dell'`ArrayList`. Per avere codice robusto, inoltre, dovremmo comunque garantirci a priori che siano inserite solo stringhe, magari con un controllo come questo:

```
 if (input instanceof String) {
     list.add(input);
 }
```

Come già osservato, senza un controllo come il precedente, potremmo andare incontro ad una delle più insidiose unchecked exception: la `ClassCastException`. Quando usiamo un tipo generico senza specificare i parametri, come abbiamo appena fatto con la classe `ArrayList`, il tipo viene detto **raw type.** Sarà utile saperlo quando leggeremo i messaggi di warning e di errore del compilatore (cfr. paragrafo 11.1.6). I generics fortunatamente ci consentono di dichiarare una lista specificando che essa accetterà solo stringhe con la seguente sintassi:

```
 List<String> strings;
```

Inoltre, bisogna anche assegnare a `strings` un'istanza che accetti lo stesso tipo di elementi, ossia `String`:

```
 List<String> strings = new ArrayList<String>();
```

che, da Java 7 in poi, possiamo scrivere direttamente così (grazie alla deduzione automatica spiegata nel paragrafo 11.1.5):

```
 List<String> strings = new ArrayList<>();
```

per evitare verbosità superflua.

Si noti che le parentesi acute <> sono solitamente chiamate "the diamond". Anche questa volta diamond dovrebbe essere tradotto come "rombo" e non come "diamante".

A questo punto abbiamo una lista che accetta solo stringhe e nessun altro tipo di oggetto. Per esempio, il seguente codice:

```
List<String> strings = new ArrayList<>();
strings.add("è possibile aggiungere String");
```

compilerà tranquillamente, mentre l'istruzione:

```
strings.add(new Date());
```

provocherà il seguente output di errore già in compilazione:

```
cannot find symbol
symbol :  method add(java.util.Date)
location: interface java.util.List<java.lang.String>
strings.add(new Date());
^
```

Possiamo immaginare che nel momento in cui viene istanziato un oggetto da una classe che implementa `List`, il compilatore rimpiazzi tutte le occorrenze di `E` con il tipo specificato tra le parentesi angolari. Infatti, per l'oggetto `strings` dell'ultimo esempio, non viene riconosciuto il metodo `add(E o)` ma `add(String o)`. Ecco spiegato il messaggio di errore. Quindi, i problemi che verranno evidenziati in fase di compilazione ne eviteranno altri ben più seri in fase di esecuzione del programma. La parametrizzazione di un tipo generico avviene non per l'intera classe, ma solo per quella particolare istanza trattata. Quindi è possibile creare più liste con differenti parametri. Come nel seguente esempio:

```
List<String> strings = new ArrayList<>();
List<Integer> ints = new ArrayList<>();
List<Date> dates = new ArrayList<>();
```

In generale tutte le classi e le interfacce come `Map`, `Iterator`, `Set`, etc. del framework Collections supportano i generics.

11.1.2 Tipi primitivi e autoboxing-autounboxing

I generici non si possono applicare ai tipi di dati primitivi. Quindi il seguente codice:

```
  List<int> ints = new ArrayList<int>();
```

restituirà il seguente messaggio di errore:

```
found :  int
required: reference
        List<int> ints = new ArrayList<int>( );
           ^
```

Fortunatamente però, la seguente sintassi:

```
  List<Integer> ints = new ArrayList<Integer>();
```

consentirà tranquillamente di aggiungere interi primitivi. Infatti in Java, grazie alla doppia feature di autoboxing-autounboxing, si possono usare in maniera intercambiabile i tipi primitivi e i relativi *tipi wrapper*. Per esempio è possibile scrivere:

```
  Integer integer = 1;
  Double d = 3.0D;
  Boolean booleano = true;
```

ma anche:

```
  char c = new Character('c');
  byte b = new Byte((byte)1);
```

oppure eseguire operazioni come:

```
  int i = 29;
  Short s = new Short ((short)7);
  Float f = 74.0F;
  double d2 = i*f-7;
```

Approfondiremo il discorso sia sull'autoboxing-autounboxing, sia sulle classi wrapper nel capitolo 12 relativo alla libreria `java.lang`.

Per tale ragione è lecito scrivere:

```
ints.add(1);
```

ed anche:

```
int intero = ints.get(0);
```

> **Si noti che le classi** `Byte`, `Short`, `Integer`, `Float`, `Double` **e** `Long` **(ed altre come** `BigDecimal`**) estendono tutte la classe astratta** `Number`.

11.1.3 Interfaccia `iterator`

Oltre a `List`, tutte le classi e tutte le interfacce del framework Collections supportano i tipi generici. Più o meno quanto visto per `List` vale per tutte le altre collection ed in particolare per `Iterator`. `Iterator` è un'interfaccia molto utilizzata del framework collection che permette di iterare su una collezione. Solitamente la chiamata al metodo `iterator()` sulla collezione utilizzata, ne restituisce un'implementazione concreta. Astrae il concetto di **iteratore:** un cursore che seleziona in maniera sequenziale gli elementi della collezione. Il metodo `hasNext()` restituisce un booleano, a seconda del fatto che ci siano ancora elementi da processare nella collezione. Il metodo `next()` invece restituisce il prossimo oggetto selezionato dall'iteratore. Per esempio, il seguente codice:

```
List<String> strings = new ArrayList<>();
strings.add("Autoboxing & Auto-Unboxing");
strings.add("Generics");
strings.add("Static imports");
strings.add("Enhanced for loop");
//...
Iterator i = strings.iterator();
while (i.hasNext()) {
    String string = i.next();
    System.out.println(string);
}
```

produrrà l'output di errore:

```
found :  java.lang.Object
required: java.lang.String
            String string = i.next();
                                    ^
1 error
```

Il problema è che bisogna dichiarare anche l'`Iterator` come generico nel seguente modo:

```
 Iterator<String> i = strings.iterator();
 while (i.hasNext()){
     String string = i.next();
     System.out.println(string);
 }
```

> **Attenzione a non utilizzare `Iterator` come generico su una `Collection` non generica. Si rischia un'eccezione al runtime se la `Collection` non è stata riempita come ci si aspetta. D'altronde i generics dovrebbero proprio prevenire questi tipi di problemi per segnalarli in fase di compilazione.**

11.1.4 Interfaccia `Map`

Una `Map` è una collezione che associa chiavi ai suoi elementi. Le mappe non possono contenere chiavi duplicate ed ogni chiave può essere associata ad un solo valore. Le mappe sono molto diverse dalle liste. Non sono ordinate come le liste, ma consentono un accesso in lettura (grazie alla chiave) più veloce rispetto ad una lista. L'interfaccia `Map` dichiara due tipi parametro (che vengono separati da una virgola). Segue la sua dichiarazione:

```
 public interface Map<K, V>
```

Questa volta i due parametri si chiamano `K` e `V`, rispettivamente iniziali di "Key" ("chiave" in italiano) e "Value" ("valore" in italiano). Infatti, per la mappa possono essere parametrizzati sia le chiavi che i valori. La più usata implementazione dell'interfaccia `Map` è la classe `HashMap`. Segue un frammento di codice che dichiara una mappa generica, con parametro chiave di tipo `Integer` e parametro valore di tipo `String`:

```
Map<Integer, String> map = new HashMap<Integer, String>();
```

Grazie all'autoboxing e all'autounboxing sarà possibile utilizzare interi primitivi per valorizzare la chiave. Per esempio, il seguente codice:

```
map.put(0, "generics");
map.put(1, "metadata");
map.put(2, "enums");
map.put(3, "varargs");
for (int i = 0; i < 4; {
    System.out.println(map.get(i));
}
```

inizializza la mappa e ne stampa i valori con un ciclo sulle sue chiavi.

È possibile anche creare innesti di generici; per esempio il seguente codice è valido:

```
Map<Integer, ArrayList<String>> map =
new HashMap<>();
```

Si tratta di una mappa che ad una chiave di tipo intero associa un arraylist di stringhe. Per ricavare un elemento dell'arraylist innestato è possibile utilizzare il seguente codice:

```
String s = map.get(chiave).get(numero);
```

dove sia `chiave` **sia** `numero` **sono interi. Possiamo notare come non abbiamo utilizzato neanche un cast.**

11.1.5 Deduzione automatica del tipo

Java 7 introdusse un piccolo cambiamento di sintassi per la creazione dei tipi generici. La **type inference for generic instance creation** è una caratteristica che potremmo tradurre come **deduzione automatica del tipo per la creazione di una istanza generica**, ma da ora in poi la chiameremo solamente **deduzione automatica (type inference).** Dal nome sembrerebbe

che si tratti di qualcosa di molto complicato, in realtà è un argomento semplice, a cui tra l'altro, abbiamo già accennato. Come tutti i cambiamenti che influenzarono la sintassi di Java 7, anche questo fu proposto dagli utenti Java e rientrò nel cosiddetto progetto "Project Coin" insieme a tutti gli altri cambiamenti che ci sono stati nella sintassi del linguaggio. Con Project Coin, Java si prefissò di semplificare la vita allo sviluppatore: scrivere meno codice per sbagliare meno. La tendenza di rendere Java meno verboso grazie alle deduzioni automatiche del compilatore, non si è poi spenta; per esempio in Java 8 sono stati introdotti argomenti come le espressioni lambda. Ma cos'è la deduzione automatica? Facciamo un semplicissimo esempio, consideriamo la seguente dichiarazione:

```
ArrayList<String> arrayList;
```

Per istanziare questo `ArrayList`, prima dell'avvento di Java 7 eravamo costretti a scrivere:

```
ArrayList<String> arrayList = new ArrayList<String>();
```

ora invece è possibile utilizzare l**'operatore diamond**, ovvero è possibile omettere i parametri per l'oggetto istanziato come nel seguente esempio:

```
ArrayList<String> arrayList = new ArrayList<>();
```

il compilatore infatti è capace di dedurre il tipo di parametro per l'istanza dell'oggetto, perché può leggere il reference a cui viene assegnato. Quindi, siccome il reference `arrayList` è stato dichiarato come `ArrayList<String>`, allora anche l'oggetto assegnato sarà dello stesso tipo. La coppia di parentesi acute vuote viene appunto detta "the diamond". Un altro esempio più significativo potrebbe essere:

```
Map<String, Set<String>> hashMap = new HashMap<String,
Set<String>>();
```

che può ora essere riscritto come:

```
Map<String, Set<String>> hashMap = new HashMap<>();
```

Si noti che possiamo utilizzare l'operatore diamond anche quando si invoca un metodo. Per esempio il seguente codice:

```
ArrayList<Number> arrayList = new ArrayList<>();
 arrayList.addAll(new ArrayList<>());
```

compila correttamente.

Sino alla versione 8 di Java, per una limitazione alla rappresentazione delle classi anonime da parte della JVM, non era possibile utilizzare la notazione diamond quando si definivano classi anonime.

In Java 9 questa limitazione è stata superata, ed ora è possibile utilizzare l'operatore diamond anche quando si definisce una classe anonima. Per esempio la classe `DiamondAnonymousClass` dichiara una classe anonima che estende `ArrayList` e riscrive il metodo `add()` in modo tale che non possano essere aggiunti duplicati:

```
public class DiamondAnonymousClass {
    List<Integer> lista = new ArrayList<>() {
        @Override
        public boolean add(Integer i) {
            if (contains(i)) {
                return false;
            }
            return super.add(i);
        }
    };
    //...
}
```

11.1.6 Raw type e XLint

Abbiamo visto come la deduzione automatica sia possibile grazie all'assegnazione di un oggetto ad un reference. Il compilatore infatti utilizzerà il parametro dichiarato dal reference come parametro dell'oggetto assegnato. Ma se il reference non dichiara un parametro, allora la deduzione automatica non potrà aver luogo. In particolare abbiamo affermato che quando si utilizza un tipo generico senza specificare il parametro, allora si parla di **raw type** (che potremmo tradurre con "tipo diretto"). Abbiamo già evidenziato che l'utilizzo di raw type, darà luogo a warning in fase di

compilazione. Per esempio, il seguente codice:

```
List strings = new ArrayList();
strings.add("Lambda");
strings.add("Streams API");
strings.add("Date and Time API");
strings.add("JavaFX");
Iterator i = strings.iterator();
while (i.hasNext()) {
    String string = (String)i.next();
    System.out.println(string);
}
```

avrà un esito di compilazione positivo, ma provocherà il seguente output:

```
Note: Generics1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Tali **warning** (o **note**, o **lint** come vengono definiti nelle specifiche) avvertono lo sviluppatore che esistono operazioni non sicure o non controllate, e viene richiesto di ricompilare il file con l'opzione `-Xlint` (che come abbiamo visto nel paragrafo 9.5.1 equivale all'opzione `-Xlint:all`), per avere ulteriori dettagli. Seguendo il suggerimento del compilatore, ricompiliamo il file con l'opzione richiesta. L'output sarà:

```
D:\javac -Xlint Generics1.java

D:\Generics1.java:5: warning: [rawtypes] found raw type: List
        List strings = new ArrayList();
        ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
D:\Generics1.java:5: warning: [rawtypes] found raw type:
ArrayList
        List strings = new ArrayList();
                           ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
D:\Generics1.java:6: warning: [unchecked] unchecked call to
add(E)
  as a member of the raw type List
        strings.add("Lambda");
```

```
                  ^
  where E is a type-variable:
    E extends Object declared in interface List
D:\Generics1.java:7: warning: [unchecked] unchecked call to
add(E)
  as a member of the raw type List
        strings.add("Streams API");
                  ^
  where E is a type-variable:
    E extends Object declared in interface List
D:\Generics1.java:8: warning: [unchecked] unchecked call to
add(E)
  as a member of the raw type List
        strings.add("Date and Time API");
                  ^
  where E is a type-variable:
    E extends Object declared in interface List
D:\Generics1.java:9: warning: [unchecked] unchecked call to
add(E)
  as a member of the raw type List
        strings.add("JavaFX");
                  ^
  where E is a type-variable:
    E extends Object declared in interface List
D:\Generics1.java:10: warning: [rawtypes] found raw type:
Iterator
        Iterator i = strings.iterator();
        ^
  missing type arguments for generic class Iterator<E>
  where E is a type-variable:
    E extends Object declared in interface Iterator
8 warnings
```

Questa volta vengono segnalate come warning le righe di codice incriminate, con una breve spiegazione nel solito stile Java. Come è possibile notare questi warning sono di due tipi: `rawtypes` e `unchecked`. Esistono infatti vari tipi di warning, di cui gli `unchecked` sono considerati i più rilevanti. Infatti, possiamo ottenere una sinossi di tutte le opzioni non standard del compilatore (ovvero che potrebbero cambiare in futuro) eseguendo il comando:

```
javac -X
```

Scopriremo che riguardo a `Xlint` esistono le seguenti tipologie:

-Xlint:<key>(,<key>)*

Warnings to enable or disable, separated by comma.
Precede a key by - to disable the specified warning.
Supported keys are:

all	Enable all warnings
auxiliaryclass	Warn about an auxiliary class that is hidden in a source file, and is used from other files.
cast	Warn about use of unnecessary casts.
classfile	Warn about issues related to classfile contents.
deprecation dep-ann	Warn about use of deprecated items. Warn about items marked as deprecated in JavaDoc but not using the @Deprecated annotation.
divzero	Warn about division by constant integer 0.
empty	Warn about empty statement after if.
exports	Warn about issues regarding module exports.
fallthrough	Warn about falling through from one case of a switch statement to the next.
finally	Warn about finally clauses that do not terminate normally.
module	Warn about module system related issues.
opens	Warn about issues regarding module

	opens.
options	Warn about issues relating to use of command line options.
overloads	Warn about issues regarding method overloads.
overrides	Warn about issues regarding method overrides.
path	Warn about invalid path elements on the command line.
processing	Warn about issues regarding annotation processing.
rawtypes	Warn about use of raw types.
removal	Warn about use of API that has been marked for removal.
requires-automatic	Warn about use of automatic modules in the requires clauses.
requires-transitive-automatic	Warn about automatic modules in requires transitive.
serial	Warn about Serializable classes that do not provide a serial version ID.Also warn about access to non-public members from a serializable element.
static	Warn about accessing a static member using an instance.
try	Warn about issues relating to use of try blocks (i.e. try-with-resources).

```
unchecked                       Warn about unchecked operations.

varargs                         Warn about potentially unsafe varargs
                                methods

none                            Disable all warnings
```

Quindi, se compilando il nostro file con l'opzione `-Xlint` otteniamo troppi warning, possiamo filtrare la lista dei warning per la tipologia che ci interessa, specificando una sotto-opzione. Per esempio specificando `-Xlint:unchecked` vedremo elencati solo i warning di tipo `unchecked`.

> **La lista dei potenziali warning include anche altre tipologie non legate obbligatoriamente a tipi generici, come già avevamo notato nel paragrafo dedicato ai warning 9.5.1.**

È anche possibile evitare di ricevere warning in compilazione, nel caso si ritenga che sia giusto non utilizzare parametri per i tipi generici. Si potrebbe utilizzare un'annotazione standard chiamata `SuppressWarnings` che dettaglieremo nel capitolo 13, relativo proprio alle annotazioni. La migliore soluzione però, è applicare la parametrizzazione ovunque sia possibile.

> **Quanto detto in questo paragrafo non si applica solo alle collection, ma a tutti i tipi generici, anche a quelli creati da noi.**

11.2 Ereditarietà e tipi generici

Anche per i tipi generici è possibile sfruttare l'ereditarietà. Quando parliamo di ereditarietà, parliamo anche del polimorfismo per dati, che altro non è che una sua conseguenza. Infatti le classi del framework Collections sono generiche e implementano interfacce generiche oppure estendono altre classi generiche. Per esempio, abbiamo utilizzato il polimorfismo per dati negli

esempi precedenti:

```
List<String> strings = new ArrayList<>();
```

Infatti `List` è un'interfaccia implementata da `ArrayList`. Il tipo parametro però, dovrà essere lo stesso. Infatti nei prossimi due paragrafi vedremo perché le gerarchie di classi non considerino valida una gerarchia tra i tipi parametro ai fini di implementare l'ereditarietà e il polimorfismo per dati.

11.2.1 Ereditarietà e tipo parametro

Anche i tipi generici formano gerarchie ma tali gerarchie non si basano sui tipi parametri dichiarati. Per esempio il seguente codice è valido:

```
ArrayList <Integer> arrayList = new ArrayList <>();
List <Integer> list = arrayList;
```

il seguente codice, invece, non è valido:

```
ArrayList <Number> list = arrayList;
```

Infatti, che `Number` sia superclasse di `Integer` non autorizza a considerare il tipo generico `ArrayList<Number>` superclasse di `ArrayList<Integer>`. Di conseguenza non è legale neanche la seguente istruzione:

```
ArrayList<Number> list = new ArrayList<Integer>;
```

Il perché sia improponibile che il tipo parametro sia la base di una gerarchia, diventa evidente con un esempio. Supponiamo di avere un `ArrayList` di `Integer`:

```
List<Integer> ints = new ArrayList<Integer>();
```

Se l'ereditarietà dipendesse anche dal tipo parametro potremmo assegnare un reference di tipo `List<Object>` ad un oggetto di tipo `List<Integer>`, siccome `Integer` è sottoclasse di `Object`, e potremmo scrivere:

```
List<Object> objs = ints;
```

ma utilizzando JShell, possiamo verificare che ci verrà restituito un messaggio di errore:

```
jshell> List<Integer> ints = new ArrayList<Integer>();
ints ==> []

jshell> List<Object> objs = ints;
|  Error:
|  incompatible types: java.util.List<java.lang.Integer>
cannot be
     converted to java.util.List<java.lang.Object>
|  List<Object> objs = ints;
|                      ^--^
```

Infatti se fosse legale assegnare ad un reference di tipo `List<Object>` un oggetto di tipo `List<Integer>`, allora sarebbe anche legale scrivere:

```
objs.add("Stringa in un generic di Integer?");
```

Ma in questo modo avremmo aggiunto una stringa in un tipo parametrizzato con `Integer`, e che quindi dovrebbe contenere solo interi! Sembra evidente che l'ereditarietà (e di conseguenza il polimorfismo per dati) non può essere determinata dal tipo parametro.

11.2.2 Type erasure

La ragione tecnica per cui l'ereditarietà non si basa sul tipo generico, è basata sul concetto di **type erasure** (in italiano **cancellazione del tipo)** a cui ci si riferisce solitamente solo con **erasure.** La gestione dei tipi generici, è gestita solo a livello di compilazione. Ricordiamo che i generics (insieme a tante altre feature come le enumerazioni, le annotazioni, il ciclo for migliorato, i varargs, etc.) sono state introdotte nella versione 5 di Java, cercando di impattare il meno possibile su quanto c'era prima. Per rendere possibile l'introduzione dei generics è stato assegnato un ruolo supplementare al compilatore. È il compilatore infatti a trasformare il codice Java in codice pre-Java 5 prima di trasformarlo in bytecode. Quindi, durante l'esecuzione, le istruzioni:

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
ArrayList<Number> list = arrayList;
```

saranno lette dalla JVM come se fossero stati cancellati (da qui il termine "erasure") i tipi generici:

```
ArrayList arrayList = new ArrayList();
ArrayList list = arrayList;
```

ma a questo punto non si potrebbero più avere informazioni sulla compatibilità degli elementi dei due vettori. È per questo che il compilatore non permette una ereditarietà basata sui parametri: al runtime i parametri non esistono più.

> **L'erasure è una delle conseguenze da pagare per la filosofia radicata di Java di essere sempre compatibile con le versioni precedenti. L'obiettivo è permettere agli sviluppatori di poter migrare senza dover buttare tutto il codice scritto in precedenza. Questo non è sempre possibile al 100%. Quindi quando si migra da una versione di Java a quella successiva, a volte è necessario ritoccare qualche dettaglio del proprio codice.**

11.2.3 Wildcard

Alla luce di quanto appena visto bisogna fermarsi un attimo per riflettere su quali possono essere le conseguenze dell'erasure. Supponiamo di avere il seguente metodo che fa uso di un raw type:

```
public void printList(ArrayList al) {
    Iterator i = al.iterator();
    while (i.hasNext()) {
        Object o = i.next();
        System.out.println(o);
    }
}
```

La compilazione di questo codice andrà a buon fine ma verranno segnalati dei warning. Nonostante sia possibile disabilitare i warning con un'opzione di compilazione (opzione `-Xlint:none` del comando `javac` cfr. paragrafo 9.5.1), sarebbe sicuramente meglio evitare che ci siano, piuttosto che sopprimerli. La soluzione più semplice sarebbe la seguente:

```
public void printList(ArrayList<Object> al) {
```

```
        Iterator<Object> i = al.iterator();
        while (i.hasNext()) {
            Object o = i.next();
            System.out.println(o);
        }
    }
```

Purtroppo però, l'utilizzo del tipo generico, per il fenomeno dell'erasure implicherà semplicemente che questo metodo accetterà in input solo tipi generici con parametro `Object` (e non per esempio `String`). Quindi l'istruzione:

```
    printList(new ArrayList<String>());
```

produrrà il seguente errore di compilazione:

```
TestWildCardError.java:25: error: cannot find symbol
        twd.printList(list);
            ^
  symbol: method printList(ArrayList<String>)
  location: variable twd of type TestWildCard
1 error
```

Quello che voleva essere un parametro polimorfo, si è rivelato quindi un errore di programmazione. Ma allora se vogliamo effettivamente decidere al volo il tipo parametro dell'arraylist da passare al metodo `printList()`, cosa si può fare? A tale scopo esiste una sintassi speciale per i generici che fa uso di **wildcard** (caratteri jolly), rappresentati da punti interrogativi. In particolare stiamo parlando di **unbounded wildcard** (letteralmente "caratteri jolly illimitati" in contrasto con le bounded wildcard che vedremo nel paragrafo 11.3.2). Il seguente codice rappresenta l'unica reale soluzione al problema presentato:

```
    public void printList(ArrayList<?> al) {
        Iterator<?> i = al.iterator();
        while (i.hasNext()) {
            Object o = i.next();
            System.out.println(o);
        }
    }
```

Facendo uso di generici, tale metodo non genererà nessun tipo di warning in

fase di compilazione ed accetterà qualsiasi tipo di parametro per l'arraylist in input. Siccome il compilatore non può controllare la correttezza del tipo di parametro quando viene utilizzata una wildcard, rifiuterà di compilare qualsiasi istruzione che tenti di aggiungere o impostare elementi nell'arraylist. Quindi, l'utilizzo delle wildcard trasforma i tipi generici *in sola lettura*. Questo significa che all'interno di questo metodo non è possibile modificare l'arraylist. Per esempio aggiungendo questa istruzione all'interno del metodo:

```
al.add("");
```

otterremo un errore in compilazione. Questo fenomeno è noto come **wildcard capture**, e può essere superato grazie a stratagemmi basati su metodi helper, come vedremo nel paragrafo 11.3.4.

11.3 Creare propri tipi generici

Come abbiamo già visto è possibile definire i propri tipi generici, come nel seguente esempio:

```
public class OwnGeneric<E> {
    private List<E> list;
    public OwnGeneric() {
        list = new ArrayList<E>();
    }
    public void add(E e) {
        list.add(e);
    }
    public void remove(int i) {
        list.remove(i);
    }
    public E get(int i) {
        return list.get(i);
    }
    public int size() {
        return list.size();
    }
```

```
    public boolean isEmpty() {
        return list.size() == 0;
    }
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        int size = size();
        for (int i = 0; i < size; i++) {
             sb.append(get(i) + (i != (size - 1) ? "-" :
"""));
        }
        return sb.toString();
    }
}
```

Probabilmente l'implementazione del metodo `toString()` ha bisogno di qualche osservazione supplementare. Per prima cosa notiamo l'utilizzo della classe `StringBuilder`. Si tratta di una classe particolarmente performante che ha dei metodi per concatenare e fare altre operazioni con le stringhe. Inoltre, l'implementazione del ciclo `for` può risultare criptica a prima vista. In effetti, l'utilizzo dell'operatore ternario non aiuta la leggibilità. Per prima cosa è possibile notare che all'interno del ciclo `for` c'è un'unica espressione. Questa aggiunge una stringa all'oggetto `sb` di tipo `StringBuilder` tramite il metodo `append()`. Il parametro di questo metodo è costituito da quanto restituisce il metodo `get()`, concatenato con il risultato dell'operatore ternario compreso tra parentesi tonde. Tale operatore ritornerà un trattino ("`-`") per separare i vari elementi estratti dalla collezione, se e solo se il valore di `i` è diverso dalla dimensione della collezione `-1`. In questo modo l'output risultante avrà una formattazione corretta.

Notiamo come il parametro sia stato definito con una `E` come si usa nella documentazione. Tuttavia, benché sia preferibile utilizzare un'unica lettera maiuscola per definire il parametro, è possibile utilizzare una qualsiasi parola valida per la sintassi Java. Con il seguente codice, invece, andiamo ad utilizzare il nostro tipo generico:

```
OwnGeneric<String> own = new OwnGeneric<String>();
for (int i = 0; i < 10; ++i) {
     own.add("" + (i));
}
System.out.println(own); //verrà chiamato il metodo
```

```
 toString()
```

L’output sarà:

```
0-1-2-3-4-5-6-7-8-9
```

> **Riguardo alla classe generica che abbiamo appena creato, non è possibile dichiarare statica la variabile d’istanza `list` nel seguente modo:**
>
> ```
> private static List<E> list = new
> ArrayList<E>();
> ```
>
> **Infatti, questo impedirebbe alle varie istanze della classe di utilizzare parametri differenti (visto che devono essere condivisi per la definizione di `static`).**

11.3.1 Parametri bounded

Il metodo `toString()` definito in `OwnGeneric` stampa i valori della variabile interna `list` separati da un trattino. Nell’esempio della classe `OwnGeneric`, il tipo generico era un `Integer` e il metodo `toString()` formattava i numeri interi in modo leggibile, separandoli con dei trattini (“-”). Ma potremmo anche utilizzare la classe nel seguente modo:

```
public class TestOwnGenericWithMinus {
    public static void main(String args[]) {
        OwnGeneric<String> own = new OwnGeneric<>();
        own.add("-");
        own.add("--");
        own.add("---");
        System.out.println(own);
    }
}
```

L’esecuzione di quest’ultima classe produrrà il seguente output, che con tutta probabilità, non è quello desiderato:

```
--------
```

È possibile creare propri tipi generici con parametri che specificano determinati tipi (**bounded parameters**). I bounded parameter sono utilissimi perché restringono il campo di scelta del parametro ad una gerarchia di classi, laddove utilizzare un parametro qualsiasi potrebbe portare situazioni non previste. Per esempio, se definiamo la classe `OwnGeneric` nel seguente modo:

```
 public class OwnGeneric <E extends Number> {
```

potremo utilizzare come parametri solo sottoclassi di `Number` (per esempio `Float, Integer...).`

Quindi se provassimo a istanziare un oggetto `OwnGeneric` con parametro stringa, otterremo un errore come mostrato di seguito:

```
jshell> OwnGeneric<String> own = new OwnGeneric<>();
|  Error:
|  type argument java.lang.String is not within bounds of
type-variable E
|  OwnGeneric<String> own = new OwnGeneric<>();
|             ^----^
|  Error:
|  incompatible types: cannot infer type arguments for
OwnGeneric<>
|      reason: inference variable E has incompatible bounds
|        equality constraints: java.lang.String
|        upper bounds: java.lang.Number
|  OwnGeneric<String> own = new OwnGeneric<>();
|                               ^-----------^
```

> **La maggior parte delle volte potremmo usare bounded parameter sulle classi create da noi. In questo modo ci metteremo al riparo da situazioni inattese. A volte conviene crearsi un'interfaccia (anche completamente vuota) che deve essere implementata da tutte le classi da usare come parametro bounded.**

Una diretta conseguenza dell'utilizzo dei bounded parameter, sarà quella di poter creare parametri covarianti in input ai metodi, come vedremo nel paragrafo 11.3.4.

11.3.2 Wildcard bounded

La classe `OwnGeneric` dichiarava un tipo generico `E`, e questo viene usato come parametro nei suoi metodi. Ora invece supponiamo di voler creare un metodo che prende in input come parametro un tipo generico con un parametro ristretto (bounded). Supponiamo però di trovarci all'interno di una classe non generica, e quindi di non avere un tipo parametro da poter sfruttare, come `E` nella classe `OwnGeneric`. Se non abbiamo un parametro, come possiamo restringere il tipo di parametro di un metodo? Non ci resta che utilizzare una wildcard, che però restringeremo come abbiamo fatto con i parametri nel paragrafo precedente. In questo caso parleremo di **bounded wildcard:**

```
public void print(List<? extends Number> list) {
    for (Iterator<? extends Number> i = list.iterator();
i.hasNext( ); ) {
         System.out.println(i.next());
    }
}
```

L'utilizzo della wildcard è obbligatorio se ci troviamo in una classe non generica (come la classe `OwnGeneric`) che già dichiara un parametro (nel caso di `OwnGeneric` si chiamava `E`). A questo metodo potremo passare solo liste con tipo parametro che sia sottoclasse di `Number`. In questo caso abbiamo usato quella che si dice **upper bounded wilcard** (ovvero wildcard limitata verso l'alto).

Esiste anche la possibilità di usare una wilcard con limite verso il basso (**lower bounded wildcard**) sfruttando, invece della parola chiave `extends`, la parola chiave `super`. Per esempio, il seguente metodo:

```
public static void riempiLista(List<? super Integer> list) {
     int size = list.size();
     for (int i = 1; i <= size; i++) {
         list.add(i);
     }
     System.out.println(list);
}
```

permette di riempire e stampare una lista riempita di `Integer, Number` (che però è una classe astratta e quindi non si può istanziare) o `Object`.

Considerando che i parametri di un metodo possono essere di tre tipi:

1) parametro che contiene dati da usare ("IN" parameter): esempio del metodo `print()`;

2) parametro che deve essere aggiornato ("OUT" parameter): esempio del metodo `riempiLista()`;

3) parametro che contiene dati da usare e che deve essere aggiornato.

Le linee guida per l'utilizzo delle wild card sono le seguenti:

- usare upper bounded wildcard per parametri di tipo 1;

- usare lower bounded wildcard per parametri di tipo 2;

- usare unbounded wildcard se la variabile di tipo 1 deve essere usata tramite metodi della classe `Object`;

- non usare wildcard per parametri di tipo 3.

Esiste anche una sintassi speciale che consente di non utilizzare la wildcard:

```
public <N extends Number> void print(List<N> list) {
    for (Iterator<N> i = list.iterator(); i.hasNext( ); ) {
        System.out.println(i.next());
    }
}
```

Quindi, con l'istruzione:

```
<N extends Number>
```

inserita prima del tipo di ritorno del metodo, stiamo dichiarando localmente un parametro chiamato `N`, che deve avere la caratteristica di estendere `Number`. Questo parametro sarà utilizzabile all'interno dell'intero metodo. Tale sintassi può essere preferibile quando, per esempio, all'interno del codice del metodo viene spesso utilizzato il parametro `<N>` (altrimenti siamo obbligati a scrivere più volte `<? extends Number>`). Questo è un esempio di **metodo generico.**

11.3.3 Metodi generici

I **metodi generici** non sono altro che metodi che definiscono i propri tipi parametro. La sintassi è la stessa che si usa per i tipi generici, ma la visibilità dei parametri è limitata alla definizione del metodo stesso, sia esso statico, un costruttore o un metodo ordinario. I tipi parametro si devono anteporre al tipo di ritorno del metodo (o al nome del metodo nel caso di metodo costruttore). Per esempio la seguente classe:

```
public class GenericMethod {
    public static <N extends Number> String getValue(N
number) {
        String value = number.toString();
        return value;
    }
    public static void main(String args[]) {
        String value = GenericMethod.getValue(new
Integer(25));
        System.out.println(value);
    }
}
```

dichiara il metodo generico `getValue()`, che dichiara un parametro bounded `N` ristretto a `Number` (che quindi potrebbe essere utilizzato con qualsiasi sottoclasse di `Number`). Si noti come anche il parametro `number` in input al metodo è di tipo `N` (che è visibile solo all'interno del metodo `getValue()`). Nel metodo `main()` poi, il metodo viene chiamato usando un tipo `Integer` con valore `25` (ma era possibile passargli anche direttamente il valore intero `25` per le regole dell'autoboxing-unboxing). Esistono anche metodi costruttori generici. In particolare quelli che consentono di utilizzare un parametro per un costruttore, dove questo parametro è diverso dal tipo parametro dichiarato per la classe. Anche in questo caso facciamo un esempio. Consideriamo la seguente classe:

```
public class AdvancedInference<Boolean> {
    public <E> AdvancedInference(E e) {
        System.out.println(e);
    }
    //...
}
```

Abbiamo specificato per il costruttore generico un tipo parametro `E` diverso

rispetto a quello definito per la classe (che era di tipo `Boolean`). Nelle versioni precedenti a Java 7 era già possibile scrivere:

```
AdvancedInference<Boolean> test = new
AdvancedInference<Boolean>("");
```

Infatti la sintassi `<Boolean>` rappresenta il tipo parametro della classe, mentre il valore `String` passato al costruttore andrà ad impostare il parametro `E`, che il metodo costruttore prende in input. Da Java 7 è possibile istanziare questa classe nel seguente modo:

```
AdvancedInference<Boolean> test = new AdvancedInference<>
("");
```

Infatti l'operatore diamond permetterà la deduzione del parametro della classe (`Boolean`), mentre il valore `String` passato al costruttore andrà ad impostare il parametro `E`, che il costruttore prende in input. Equivalente a questa sintassi, esiste anche la seguente:

```
AdvancedInference<Boolean> test3 = new
<String>AdvancedInference<Boolean>("");
```

nella quale si esplicita anche il parametro del tipo del costruttore `<String>`. Questa sintassi è valida ma non è usata molto spesso, la riportiamo solo per completezza di informazioni.

11.3.4 Wildcard capture e metodi helper

Nel paragrafo 11.2.3, avevamo visto che un parametro di tipo collection parametrizzato con una wildcard, rendeva la collection non modificabile all'interno del metodo. Infatti possiamo verificare velocemente con JShell, che il seguente codice non compila:

```
jshell> public class WildCardTest {
   ...>     void test(List<?> list) {
   ...>         list.set(0, list.get(0));
   ...>     }
   ...> }
|  Error:
|  incompatible types: java.lang.Object cannot be converted to
capture#1 of ?
```

```
|           list.set(0, list.get(0));
|                        ^---------^
```

Infatti `list` è parametrizzato con una wildcard, ma all'interno del codice del metodo si sta provando ad inserire un oggetto che il compilatore deduce essere un `Object`. Eppure la lista potrebbe essere parametrizzata al runtime da stringhe come nel seguente esempio:

```
List<String> param = new ArrayList();
param.add("string 1");
param.add("string 2");
param.add("string 3");
WildCardTest wct = new WildCardTest();
wct.test(param);
```

Anche se `param` è parametrizzato con stringhe, il compilatore non riesce a dedurre correttamente il tipo parametro all'interno del metodo `test()`. In particolare per la definizione stessa di wildcard, considera gli elementi della collection `Object`, e un `Object` non può essere messo in una lista che sarà parametrizzata solo al runtime (nel nostro caso con stringhe). Il compilatore ci sta dicendo che siccome la wildcard potrebbe essere inizializzata con un qualsiasi tipo, al runtime la collection potrà essere parametrizzata con tale tipo, e quindi non si può assumere a priori che possa essere inizializzata con un `Object`. Questo fenomeno di *deduzione limitata*,è nota come **wildcard capture.** Un modo per risolvere la situazione è creare un **metodo helper.** Un metodo helper è un metodo generico (cfr. paragrafo 11.3.3) che definirà un parametro per la collection da sfruttare per aggiornare la collection stessa. Per esempio la seguente implementazione è legale:

```
public class WildCardTest {
    void test(List<?> list) {
       testHelper(list);
    }
    //Metodo helper
    private <T> void testHelper(List<T> list) {
       list.set(0, list.get(0));
    }
}
```

Infatti sfruttando il parametro `T` di un metodo generico, siamo riusciti ad

aggiornare la collection che nella precedente implementazione non era possibile aggiornare. In questo caso, all'interno del metodo `testHelper()`, il compilatore utilizza il parametro `T` (e non la wildcard) che al runtime viene parametrizzato come stringa. Se in fase di esecuzione `List<T> list` è parametrizzato con una stringa, allora l'istruzione `list.get(0)` restituirà una stringa.

> **La tecnica dei metodi helper ha anche una convenzione per il nome del metodo ausiliare, che dovrebbe essere del tipo `nomeDelMetodoOriginaleHelper`, ovvero `nomeDelMetodoOriginale` con l'aggiunta di `Helper` alla fine del nome.**

Potremo sfruttare questa tecnica nel momento in cui troveremo in fase di compilazione messaggi di errore che riportano chiaramente errori di *capture*.

11.4 Parametri covarianti

Quando abbiamo parlato delle regole che governano l'override nel capitolo 8 abbiamo accennato ai **tipi di ritorno covarianti**, applicando i quali è possibile creare override di metodi il cui tipo di restituzione è una sottoclasse del tipo di restituzione del metodo originale. Per esempio, sempre considerando il rapporto di ereditarietà che sussisteva tra le classi `Punto` e `PuntoTridimensionale`, se nella classe `Punto` fosse presente il seguente metodo:

```
public Punto elaboraPunto() {
    //...
}
```

sarebbe legale implementare il seguente override nella classe `PuntoTridimensionale`:

```
@Override
public PuntoTridimensionale elaboraPunto() {
    //...
```

```
  }
```

Le regole dell'override però non ci permettono di avere anche i parametri di input di un metodo che siano covarianti. Questo significa che se nella classe `Punto` fosse presente il metodo `distanza()` che misura la distanza del punto corrente da un altro punto:

```
  public class Punto {
      public double distanza(Punto altroPunto) {
          //...
      }
  }
```

e provassimo a fare override di tale metodo nella sua sottoclasse, specificando come input invece di un `Punto` un `PuntoTridimensionale` (che definiamo **parametro covariante**):

```
  public class PuntoTridimensionale extends Punto {
      @Override
      public double distanza(PuntoTridimensionale altroPunto)
  {
          //...
      }
  }
```

otterremmo un errore in compilazione:

```
Error:
method does not override or implement a method from a
supertype
  @Override
  ^-------^
```

Infatti, per fare un override, la firma del metodo riscritto deve essere identica alla firma del metodo esteso. Nel caso non coincidano avremmo due metodi diversi nella sottoclasse, e si deve parlare di overload e non di override (cfr. capitolo 8). Non si può parlare quindi dell'esistenza di parametri covarianti in senso stretto. Con i tipi generici però, è possibile implementare uno stratagemma per usare parametri covarianti. Parliamo di uno stratagemma (o sarebbe meglio definirlo pattern) non banale. Nei prossimi paragrafi infatti studieremo l'esempio probabilmente più complesso presentato sinora.

11.4.1 Problema dei parametri covarianti

Consideriamo le due seguenti interfacce:

```
public interface Cibo {
    String getColore();
}
public interface Animale {
    void mangia(Cibo cibo);
}
```

È facile implementare l'interfaccia `Cibo` nella classe `Erba`:

```
public class Erba implements Cibo {
    public String getColore() {
         return "verde";
    }
}
```

Così come è facile implementare l'interfaccia `Animale` nella classe `Carnivoro`:

```
public class Carnivoro implements Animale {
   public void mangia(Cibo cibo) {
       //un carnivoro potrebbe mangiare erbivori
   }
}
```

E ancora, implementiamo, sia l'interfaccia `Animale` sia l'interfaccia `Cibo`, nella classe `Erbivoro` (dato che un erbivoro potrebbe essere il cibo di un carnivoro):

```
public class Erbivoro implements Cibo, Animale {
   public void mangia(Cibo cibo) {
       //un erbivoro mangia erba
   }

   public String getColore() {
       //...
   }
}
```

Il problema è che in questo modo sia un carnivoro sia un erbivoro, potrebbero mangiare qualsiasi cosa; un carnivoro potrebbe cibarsi d'erba e questo è

inverosimile. Potremmo risolvere la situazione sfruttando controlli interni ai metodi che, tramite l'operatore `instanceof` e l'eventuale lancio di un'eccezione personalizzata (`ciboException`) al runtime, impongano i giusti vincoli alle nostre classi. Per esempio:

```
public class Carnivoro implements Animale {
   public void mangia(Cibo cibo) throws CiboException {
        if (!(cibo instanceof Erbivoro)) {
            throw new CiboException("Un carnivoro deve " +
             "mangiare erbivori!");
        }
        //...
    }
}
public class Erbivoro implements Cibo, Animale {
    public void mangia(Cibo cibo) throws CiboException {
        if (!(cibo instanceof Erba)) {
            throw new CiboException("Un erbivoro deve
mangiare erba!");
        }
        //...
     }

     public String getColore() {
        //...
     }
}
public class CiboException extends Exception {
    public CiboException(String msg) {
        super(msg);
    }
    //...
}
```

Inoltre per le regole dell'override relative alle eccezioni (cfr. capitolo 9), per poter compilare correttamente le nostre classi dovremo anche ridefinire l'interfaccia `Animale` in modo tale che il metodo `mangia()` soggetto a override definisca una clausola `throws` per il lancio di una `CiboException`. Segue la ridefinizione dell'interfaccia `Animale`:

```
public interface Animale {
    void mangia(Cibo cibo) throws CiboException;
}
```

Le nostre classi sono ora astratte più correttamente, ma un eventuale problema verrà rilevato solo durante l'esecuzione dell'applicazione. Per esempio, il seguente codice compilerà correttamente salvo poi sollevare un'eccezione al runtime:

```
public class TestAnimali {
    public static void main(String[] args) {
      try {
          Carnivoro tigre = new Carnivoro();
          Cibo erba = new Erba();
          tigre.mangia(erba);
      } catch (CiboException exc) {
          exc.printStackTrace();
      }
    }
}
```

Ecco l'output:

```
CiboException: Un carnivoro deve mangiare erbivori!
    at Carnivoro.mangia(Carnivoro.java:4)
    at TestAnimali.main(TestAnimali.java:6)
```

11.4.2 Soluzione con bounded parameter

L'ideale però, sarebbe rendere più specifico il parametro polimorfo del metodo `mangia()`, definendolo di tipo `Erba` per la classe `Erbivoro`, e di tipo `Erbivoro` per la classe `Carnivoro`. In definitiva ci piacerebbe utilizzare i parametri covarianti nel metodo `mangia()`. In questo modo infatti, il codice precedente non sarebbe neanche compilabile, e questo sarebbe un vantaggio non da poco. Proviamo in primo luogo ad immaginare come dichiarare i parametri covarianti che risolverebbero i nostri problemi:

```
public class Carnivoro implements Animale {
    public void mangia(Erbivoro erbivoro) {
        //...
    }
}
public class Erbivoro implements Cibo, Animale {
    public void mangia(Erba erba) {
        //...
    }
```

```
      public String getColore() {
          //...
      }
 }
```

Purtroppo questo codice non compilerà, perché appunto non avremo implementato in nessuna delle due classi il metodo `mangia()` che prende come parametro un oggetto di tipo `Cibo`. Infatti la classe `Carnivoro` implementa un metodo `mangia()` che prende in input un oggetto di tipo `Erbivoro`, mentre la classe `Erbivoro` definisce un metodo `mangia()` che prende in input un oggetto di tipo `Erba`. Quindi, ereditando il metodo astratto `mangia()` che prende in input un oggetto di tipo `Cibo`, in classi non astratte, otterremo errori in compilazione:

```
Error:
Carnivoro is not abstract and does not override abstract
method
  mangia(Cibo) in Animale
public class Carnivoro implements Animale {
^-----------------------------------------...
Error:
Erbivoro is not abstract and does not override abstract method
  mangia(Cibo) in Animale
public class Erbivoro implements Cibo, Animale {
^----------------------------------------...
```

Per raggiungere il nostro scopo possiamo però sfruttare appunto i tipi generici. Ridefiniamo l'interfaccia `Animale`, parametrizzandola con un *bounded parameter* nel seguente modo:

```
 public interface Animale<C extends Cibo> {
     void mangia(C cibo);
 }
```

Ora possiamo ridefinire le classi `Carnivoro` ed `Erbivoro` utilizzando per la classe `Carnivoro` un bounded parameter `E` che limita i tipi parametri alle sottoclassi di `Erbivoro`, mentre per la classe `Erbivoro` un bounded parameter `E` che restringe i tipi parametri alle sottoclassi di `Erba`:

```
 public class Carnivoro<E extends Erbivoro> implements
 Animale<E> {
```

```
    public void mangia(Erbivoro erbivoro) {
        //un carnivoro potrebbe mangiare erbivori
    }
}
public class Erbivoro<E extends Erba> implements Cibo,
Animale<E> {
    public void mangia(E erba) {
        //un erbivoro mangia erba
    }
    public String getColore() {
        //...
    }
}
```

Il codice precedente viene compilato correttamente e impone i giusti vincoli per le gerarchie create. Infatti, consideriamo la seguente classe con metodo `main()` che utilizza correttamente le classi `Carnivoro, Erbivoro` e `Erba:`

```
public class TestAnimali {
    public static void main(String[] args) {
    Carnivoro<Erbivoro> tigre = new Carnivoro<Erbivoro>();
    Erbivoro <Erba> erbivoro = new Erbivoro<Erba>();
    Erba erba = new Erba();
    erbivoro.mangia(erba);
    tigre.mangia(erbivoro);
  }
}
```

Possiamo notare che l'oggetto `Tigre` viene parametrizzato con il tipo `Erbivoro.` Questo significa che al suo metodo `mangia()` è possibile passare solo oggetti di tipo `Erbivoro` (o di eventuali sottoclassi). Discorso analogo per l'oggetto `Erbivoro` che viene parametrizzato con un tipo `Erba:` questo implica che è possibile passare al suo metodo `mangia()` solo oggetti di tipo `Erba` (o di eventuali sottoclassi).

Adesso il metodo `mangia()` dell'interfaccia `Animale` non deve più definire la clausola `throws` per poter lanciare una `CiboException.` Anzi, la classe `CiboException` non è più necessaria.

Se invece provassimo a far mangiare ad un carnivoro dell'erba, questa volta otterremo un errore in compilazione e non al runtime:

```
jshell> Carnivoro<Erba> tigre = new Carnivoro<Erba>();
|  Error:
|  type argument Erba is not within bounds of type-variable E
|  Carnivoro<Erba> tigre = new Carnivoro<Erba>();
|                ^--^
|  Error:
|  type argument Erba is not within bounds of type-variable E
|  Carnivoro<Erba> tigre = new Carnivoro<Erba>();
|                                        ^--^
```

Il bounded parameter della classe `Carnivoro` infatti, ci vincola ad utilizzare solo parametri corretti. Concludendo, con i bounded parameter siamo riusciti ad implementare i parametri covarianti dei metodi!

> **Nei precedenti esempi era possibile tranquillamente utilizzare l'operatore diamond. Non l'abbiamo fatto solo per rendere più leggibili gli esempi.**

Riepilogo

I **tipi generici** sono tipi che consentono di essere utilizzati impostando un parametro al momento dell'istanza. I tipi generici definiscono solitamente metodi che utilizzano tale parametro, e il tipo del parametro può cambiare da oggetto ad oggetto. Il grande vantaggio che abbiamo è quello di poter evitare eccezioni in fase di esecuzione (bug) dovuti a `ClassCastException`. Con i generici riusciamo a comprendere eventuali problemi in fase di compilazione.

La funzionalità nota come **deduzione automatica**, ci permette di rendere il codice meno verboso, evitando di scrivere laddove superfluo il tipo parametro di un'istanza di un tipo generico. La coppia di parentesi angolari viene detta **operatore diamond.** Da Java 9 in poi è possibile utilizzare l'operatore diamond anche quando si definiscono classi anonime.

I tipi generici che vengono istanziati senza specificare i tipi parametri vengono detti **raw type**, e il loro utilizzo genera eccezioni in fase di compilazione. Con l'opzione **-Xlint** riusciamo a filtrare eventuali warning del

compilatore.

Il **framework Collections** contiene alcune delle classi più utilizzate dai programmatori Java, implementate come tipi generici. Dopo aver valutato la parametrizzazione di **liste, mappe** e **iteratori** con i generici, abbiamo cercato di capire qual è il lavoro del compilatore dietro le quinte. L**'autoboxing-unboxing** permette di utilizzare le classi wrapper come parametri dei tipi generici al posto dei tipi primitivi in maniera trasparente.

L**'ereditarietà** di tipi generici non si basa sul tipo parametro. Infatti l**'erasure**, va a cancellare il tipo parametro al momento della compilazione (questo per mantenere la compatibilità del nuovo codice con il codice pre-Java 5). Quindi il tipo parametro non può tecnicamente avere un ruolo nella definizione di ereditarietà. Con le **wildcard** possiamo generalizzare i parametri di un tipo generico. La wildcard sostituisce l'utilizzo del tipo `Object`, che per le regole dell'erasure non supporta l'ereditarietà e quindi il polimorfismo per dati. Lo si usa quando non si conosce a priori il tipo di parametro che sarà usato.

Possiamo creare anche **tipi generici personalizzati**, semplicemente definendo parametri alla destra della dichiarazione del tipo, ed eventualmente utilizzare tali parametri all'interno dei metodi definiti.

Con i **parametri bounded** possiamo limitare i parametri di un certo tipo generico, restringendo così i possibili tipi di parametri che si possono utilizzare per istanziare oggetti.

Allo stesso modo con le **wildcard bounded** possiamo limitare i parametri passati ai metodi ad essere sottoclassi di un certo tipo.

I **metodi generici** sono molto utili per specificare solo localmente il parametro ed utilizzarlo all'interno del codice del metodo.

I **metodi helper**, sono metodi di supporto per risolvere il problema della **wildcard capture**, ovvero l'impossibilità di aggiungere un elemento ad un tipo generico che ha come parametro una wildcard. Un metodo helper è un metodo generico che definisce il tipo parametro che può essere usato per modificare la collection parametrizzata con wildcard. Si passa tale collection al metodo che si occuperà di modificare la collection.

In ultimo abbiamo visto come sia possibile utilizzare **parametri di metodi covarianti** utilizzando un complesso pattern basato sui **bounded parameter.**

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere e saper utilizzare i tipi generici (unità 11.1)	☐	
Comprendere la deduzione automatica del tipo, i raw type e gestire i warning del compilatore (unità 11.1)	☐	
Comprendere l'erasure e come l'ereditarietà si applica con i tipi generici (unità 11.2)	☐	
Saper usare le wildcard e i parametri bounded (unità 11.2, 11.3, 11.4)	☐	
Saper creare i propri tipi generici (unità 11.3)	☐	
Saper creare metodi generici (unità 11.3)	☐	
Saper gestire le wildcard capture con i metodi helper (unità 11.3)	☐	
Saper creare parametri covarianti (unità 11.4)	☐	

Note:

12

La libreria indispensabile: il package `java.lang`

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Conoscere l'utilizzo e i metodi più importanti della classe `Object` (unità 12.1).
- ✔ Conoscere l'utilizzo e i metodi più importanti della classe `System` (unità 12.2).
- ✔ Conoscere l'utilizzo e i metodi più importanti della classe `Runtime` (unità 12.3).
- ✔ Conoscere l'utilizzo e i metodi più importanti della classe `Class` e saper sfruttare la tecnica della reflection (unità 12.4).
- ✔ Conoscere l'utilizzo e i metodi più importanti della classe `String` (unità 12.5).
- ✔ Comprendere l'utilizzo delle classi wrapper (unità 12.6).
- ✔ Comprendere le semplificazioni che ci offre la (doppia) funzionalità di autoboxing e autounboxing (unità 12.6).
- ✔ Conoscere l'utilizzo e i metodi più importanti della classe `Math` (unità 12.7).
- ✔ Conoscere le interfacce `Comparable` e `Comparator` per ordinare elementi di array e collezioni (unità 12.8).

Sino ad ora abbiamo parlato solo superficialmente delle librerie Java. Sono tantissime, e per coprirle tutte ci vorrebbero decine di libri come questo. In effetti non esiste al mondo alcun programmatore Java che conosca tutte le librerie, è letteralmente impossibile. È invece fondamentale avere sempre a disposizione la documentazione della libreria standard, per consultarla all'occorrenza e trovare soluzioni. Esistono però delle classi e delle interfacce che possono essere considerate basilari. Se non si conosce questo minuscolo sottoinsieme della libreria standard, non si può dire di avere una cultura Java sufficiente. Certamente bisogna conoscere almeno i meccanismi basilari di queste librerie e quando utilizzarli, facendo poi riferimento alla documentazione ufficiale per comprendere a fondo certi concetti, se non ancora chiari o perché dimenticati.

È anche possibile leggere i file sorgenti della libreria Java. Essi sono contenuti in un file chiamato src.zip, che sino alla versione 8 di Java era posizionato nella cartella di installazione del JDK. Era possibile navigare direttamente tra le cartelle che rappresentavano i package, ed esplorare le classi che ci interessavano. Nel JDK 9 il file src.zip è stato spostato nella sottocartella lib, e i package sono stati inclusi nelle cartelle che rappresentano i moduli. Per esempio il sorgente String.java si trova ora sotto le cartelle java.base/java/lang. Infatti java.base è il nome del modulo che contiene il package java.lang.

La libreria di Java 9 è quindi ora organizzata in moduli, ed ogni modulo contiene dei package. In questo libro descriveremo diverse librerie fondamentali, e la quasi totalità di esse si trova all'interno del modulo chiamato `java.base`. È arrivato quindi il momento di introdurre il package che viene utilizzato in ogni programma: il package `java.lang`. In effetti è l'unico package automaticamente importato in ogni programma Java. Il package `java.lang`, oltre a contenere la classe `Thread` e l'interfaccia `Runnable` di cui parleremo abbondantemente nel capitolo 15 dedicato ai thread, contiene altre decine di classi che già abbiamo utilizzato in questo testo. Questo capitolo non coprirà tutte le classi presenti in questo package, né lo scopo è quello di sostituire la documentazione ufficiale delle API di Java. Cercheremo piuttosto di capire come utilizzare le classi e le interfacce principali, e approfondiremo alcuni concetti particolarmente ostici. Inoltre dettaglieremo la caratteristica di Java chiamata **autoboxing-autounboxing**

che permette di utilizzare i tipi primitivi in luogo di quelli complessi e viceversa. Introdurremo infine un argomento fondamentale noto come **reflection**, definito dal sottopackage `java.lang.reflection` che è alla base di molte tecnologie basate su Java. Faremo anche qualche incursione in altri package come `java.util`, come nel caso dell'interfaccia `Comparator`, e alcune classi d'utilità come `Properties`, `Objects`, `Arrays` e `Collections`.

12.1 La classe `Object`

Iniziamo dalla classe che è alla cima della gerarchia delle classi: `Object`. Essendo superclasse di tutte le classi, i suoi undici metodi saranno sempre ereditati. Inoltre, `Object` non contiene variabili.

> **Concettualmente la classe `Object` avrebbe dovuto essere dichiarata astratta e con metodi astratti. Non definendo variabili forse poteva essere dichiarata anche come interfaccia. Ma dato che ogni classe scritta in Java estende la classe `Object`, questo avrebbe comportato che per ogni classe creata avremmo dovuto fare override di tutti i suoi metodi astratti... un linguaggio così non avrebbe attirato molti sviluppatori!**

12.1.1 Il metodo `equals()`

Un metodo davvero importante della classe `Object` è indubbiamente il metodo `equals()`. Esso è progettato per confrontare due reference (sul primo viene chiamato il metodo e il secondo viene passato come parametro) e restituisce un valore booleano `true` se e solo se i due reference puntano ad uno stesso oggetto (stesso indirizzo di puntamento). Ma questo tipo di confronto, come abbiamo avuto modo di constatare nel capitolo 3, è fattibile anche mediante l'operatore `==`. In effetti il metodo `equals()` nella classe `Object` è definito come segue:

```
public boolean equals(Object obj) {
    return (this == obj);
```

```
}
```

La classe `Object` infatti è troppo generica per poter avere un'implementazione più accurata, per cui nella maggior parte delle sottoclassi di `Object`, (per esempio `String`), il metodo `equals()` è stato riscritto (override) in modo tale da restituire `true` anche nel caso di confronto tra due reference che puntano ad oggetti diversi, ma con gli stessi contenuti. Dovrebbe essere questo il vero scopo di un metodo come `equals()`. Se vogliamo confrontare due oggetti creati da una nostra classe quindi, dovremmo effettuare l'override del metodo `equals()`, in modo tale che questo restituisca `true` se le variabili d'istanza dei due oggetti coincidono. Per esempio, un buon override del metodo `equals()` per la classe `Punto` potrebbe essere la seguente:

```
@Override
public boolean equals(Object altroOggetto) {
    if (!(altroOggetto instanceof Punto)) {
        return false;
    }
    Punto altroPunto = (Punto) altroOggetto;
    return this.x == altroPunto.x && this.y == altroPunto.y;
}
```

Inizialmente controlliamo se `altroOggetto` non è di tipo `Punto` e in tal caso ritorniamo `false`. L'operatore `instanceOf` restituisce `false` anche se il primo operando è un reference `null`. Poi castiamo `altroOggetto` di tipo `Object` ad `altroPunto` di tipo `Punto` e infine controlliamo l'effettiva uguaglianza di entrambe le variabili d'istanza `x` e `y` dei due oggetti sfruttando l'operatore `&&`.

12.1.2 Il metodo `hashcode()`

C'è un metodo che è strettamente legato al metodo `equals()` : il metodo `hashcode()`. Infatti se facciamo override del metodo `equals()`, dovremmo anche fare override di `hashcode()` e viceversa. In particolare compilando l'esempio precedente con l'opzione -Xlint otterremo il seguente warning:

```
javac Punto.java
Punto.java:1: warning: [overrides] Class Punto overrides
```

```
equals,
  but neither it nor any superclass overrides hashCode
method
public class Punto {
       ^
```

Il metodo `hashcode()` viene utilizzato silenziosamente da alcune classi molto importanti (come `HashMap`) del framework Collections. Esso ha un ruolo complementare al metodo `equals()` per confrontare l'uguaglianza di due oggetti presenti nella stessa collezione. Un **hash code** (in italiano spesso chiamato **codice hash)** non è altro che un `int` che rappresenta un oggetto. Nella documentazione del metodo `hashCode()` della classe `java.lang.Object`, viene definito il cosiddetto **contratto** di `hashCode()`, che definisce alcune regole per l'implementazione di questo metodo.

1. Se due oggetti sono uguali relativamente al loro metodo `equals()`, allora devono avere anche lo stesso hash code. Quindi, dati due oggetti `o1` e `o2`, se `o1.equals(o2)` restituisce `true`, allora tali oggetti devono avere anche lo stesso hash code.
2. Il viceversa della regola 1) non fa parte del contratto: due oggetti con lo stesso hash code non devono essere obbligatoriamente uguali relativamente al loro metodo `equals()`. Questo significa che se `o1.hashcode()` è uguale a `o2.hashcode()`, non è necessario che `o1.equals(o2)` restituisca `true`.
3. Se il metodo `hashcode()` è chiamato più volte all'interno della stessa esecuzione, il metodo deve ritornare sempre lo stesso valore intero.
4. Per calcolare l'hash code, il metodo `hashcode()` deve utilizzare variabili d'istanza che sono utilizzate anche nel metodo `equals()`.

Per esempio, un override del metodo `hashCode()` per la classe `Punto` potrebbe essere:

```
public int hashCode() {
     return x + y;
}
```

La creazione di un buon metodo `hashcode()` potrebbe anche essere molto più

complessa. Per esempio, il seguente metodo è sicuramente più efficiente del precedente per quanto riguarda sia la precisione che le prestazioni:

```
public int hashCode() {
    return x ^ y;
}
```

Altri metodi potrebbero risultare più accurati, rendendo minore la probabilità di errore (ovvero di collisioni di hash code tra più oggetti), ma bisogna tenere conto delle prestazioni. Infatti, in alcune situazioni una collezione potrebbe invocare in maniera insistente il metodo `hashCode()`. Un buon algoritmo di hash dovrebbe costituire un compromesso accettabile tra precisione e performance, ma la sua implementazione dipende anche dalle esigenze dell'applicazione. Nonostante la materia degli hash code polinomiali sia vasta e complessa, una convinzione comune (ma non di tutti) è quella che un buon algoritmo di hash coinvolga la moltiplicazione per un numero primo come 31 (di solito insieme a questa moltiplicazione si usano anche altre operazioni). Per esempio un algoritmo considerato in generale migliore del precedente è il seguente:

```
public int hashCode() {
    return 31 * (x + y);
}
```

In generale, IDE evoluti come Eclipse e Netbeans permettono la generazione automatica dei metodi `equals()` e `hashCode()`. Eclipse usa proprio una moltiplicazione per 31, mentre Netbeans usa un algoritmo più complesso che coinvolge più di un numero primo.

È importante comunque tenere presente queste osservazioni per non perdere tempo in debug evitabili.

Supponiamo ora che `x` e `y` siano stati definiti come tipi complessi, per esempio come stringhe. Allora potremmo riscrivere il metodo `hashCode()` nel seguente modo:

```
```

```
public int hashCode() {
    final int prime = 31;
    int result = prime * (((x == null) ? 0 :  x.hashCode())
+
                          ((y == null) ? 0 :
y.hashCode()));
    return result;
}
```

Si noti che abbiamo introdotto controlli di nullità mediante l'operatore ternario a scapito però della leggibilità del codice. In casi come questo ci può essere d'aiuto la classe `Objects`, una classe di utilità appartenente sempre al package `java.lang` che contiene solo metodi statici. Infatti al suo interno troviamo il metodo statico `hashCode()` che prende in input un `Object`, restituendo l'hash code dell'oggetto passato in input, oppure `0` se l'oggetto è `null`. Quindi possiamo riscrivere il metodo precedente nel seguente modo:

```
public int hashCode() {
    return 31 * (Objects.hashCode(x) + Objects.hashCode(y));
}
```

All'interno di `Objects` troviamo anche il metodo `hash()` che prende in input un varargs di `Object`, e quindi potremmo anche scrivere:

```
public int hashCode() {
    return 31 * Objects.hash(x, y);
}
```

12.1.3 Il metodo `toString()`

Il metodo `toString()` ha il compito di restituire una stringa descrittrice dell'oggetto. Nella classe `Object`, che astrae il concetto di oggetto generico, tale metodo non poteva adempiere a questo scopo, giacché un'istanza della classe `Object` non ha variabili d'istanza che la caratterizzano. I progettisti di Java decisero quindi di implementare il metodo `toString()` in modo tale che questo restituisse una stringa contenente informazioni sul reference. Infatti tale metodo restituisce una stringa nel formato: `NomeClasse@IndirizzoInFormatoEsadecimale`, che per la convenzione definita in precedenza potremmo interpretare come `IntervalloDiPuntamento@IndirizzoInFormatoEsadecimale`. In particolare

il metodo `toString()`, è implementato nel seguente modo:

```
public String toString() {
    return getClass().getName() + "@" +
Integer.toHexString(hashCode());
}
```

Ovvero viene chiamato il metodo `getClass()`, che restituisce un oggetto `Class`, e su questo viene a sua volta chiamato il metodo `getName()` che restituisce il nome della classe. Poi si concatena la stringa `@`, e dopo si invoca il metodo `hashcode()` che restituisce l'indirizzo in memoria dell'oggetto. Questo però viene tramutato in un numero esadecimale perché passato in input al metodo statico `toHexString()` della classe `Integer`. Per esempio consideriamo:

```
public class ToStringDemo {
    public static void main(String args[]) {
        ToStringDemo object = new ToStringDemo();
        System.out.println(object);
    }
}
```

Nel metodo `main()` la classe `ToStringDemo` istanzia sé stessa per essere stampata. Nel codice è implicito che la classe `ToStringDemo` estende `Object` e che quindi eredita il metodo `toString()`, e che questo viene chiamato sul reference `object`, una volta passato come input al metodo `println()`. Infatti l'output di questo programma è il seguente:

```
ToStringDemo@5679c6c6
```

Il metodo `toString()` è solitamente implementato nelle sottoclassi allo scopo di restituire i valori delle variabili d'istanza delle sottoclassi. Un valido aiuto è offerto dai tool di sviluppo che automatizzano l'implementazione di questo metodo, permettendo allo sviluppatore di scegliere quali campi includere nella stringa restituita.

12.1.4 Il metodo `clone()`

Infine, la classe `Object` definisce un metodo `clone()` che per contratto restituisce una copia dell'oggetto corrente. La clonazione produce un oggetto

della stessa classe dell'originale con le variabili d'istanza contenenti gli stessi valori, e su cui sono permesse modifiche senza che l'oggetto originario sia modificato. A volte clonare un oggetto ci consente di lavorare su una copia senza il rischio di creare qualche problema all'originale. Per garantire che il metodo `clone()` non sia invocato accidentalmente, il metodo è dichiarato `protected`. Essendo dichiarato `protected`, per essere reso disponibile per l'invocazione nelle nostre classi, bisognerà sottoporlo a override cambiandone il modificatore a `public`, come nella seguente classe di esempio:

```
 public class CloneableClass implements Cloneable {
     private String campo;
     // Dichiarazione dei metodi set e get omessi...
     @Override
     public Object clone() throws CloneNotSupportedException
 {
         return super.clone();
         }

         @Override
         public String toString() {
             return " Oggetto " + super.toString() + " - " +
 getCampo();
         }
 }
```

Si noti che abbiamo dovuto anche dichiarare la clausola `throws CloneNotSupportedException`, e la nostra classe ha dovuto implementare l'interfaccia `Cloneable`. Infatti, nel caso si chiamasse il metodo `clone()` su un oggetto la cui classe non implementi la suddetta interfaccia, il metodo `clone()` della stessa classe `Object`, lancerebbe proprio una `CloneNotSupportedException` nel momento in cui fosse chiamato il metodo `clone()`, producendo il seguente output:

```
Exception in thread "main"
java.lang.CloneNotSupportedException:
  CloneableClass
      at java.lang.Object.clone(Native Method)
      at CloneableClass.clone(CloneableClass.java:15)
      at TestCloneableClass.main(TestCloneableClass.java:6)
```

Tutto questo sempre per garantire che il metodo `clone()` non sia invocato accidentalmente. Quindi le seguenti righe:

```
CloneableClass cloneableClass = new CloneableClass();
cloneableClass.setCampo("TEST");
System.out.println(cloneableClass);
System.out.println(cloneableClass.clone());
```

produrranno il seguente output:

```
Oggetto CloneableClass@15db9742 - TEST
Oggetto CloneableClass@6d06d69c - TEST
```

È importante notare che la copia che viene fatta dal metodo `clone()` della classe `Object`, si limita a copiare i valori delle variabili d'istanza dell'oggetto (si parla infatti di **shallow copy** ovvero di **copia superficiale).** Quindi se le variabili d'istanza sono reference ad altri oggetti, allora verranno copiati i loro indirizzi, con il risultato che l'oggetto clonato non clonerà anche gli oggetti variabile d'istanza dell'oggetto originale, ma questi saranno condivisi come mostrato in figura 12.1. Questo tipo di situazione non è desiderabile per una clonazione. Per effettuare una **deep copy** (in italiano **copia profonda)**, bisogna riscrivere il metodo `clone()`, evitando di chiamare il metodo di `Object` e clonando uno ad uno anche tutte le variabili d'istanza di tipo complesso. Una soluzione percorribile potrebbe anche essere quella di implementare il metodo `clone()` nelle classi delle variabili d'istanza complesse, ma questo non eviterebbe che il problema si riproponga anche su queste classi, che a loro volta potrebbero dichiarare al proprio interno altri attributi complessi.

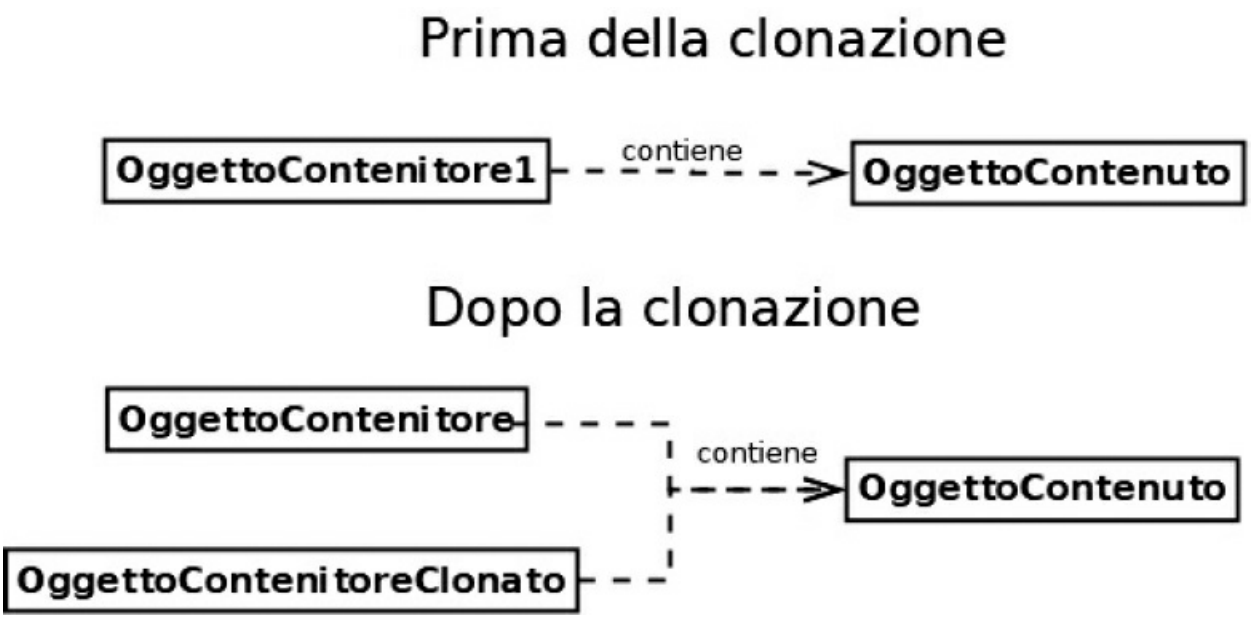


Figura 12.1 - Shallow copy.

> **Nel caso si voglia clonare un oggetto definendo un altro metodo e non si invochi il metodo `clone()` della superclasse `Object`, allora non sarà più necessario dichiarare la clausola `throws CloneNotSupportedException`. È comunque buona norma implementare l'interfaccia `Cloneable`.**

Esistono altri metodi importanti nella classe `Object`, li incontreremo proseguendo nella lettura di questo testo.

12.2 La classe `System`

La classe `System` astrae il sistema su cui si esegue il programma Java. Non è instanziabile in quanto dichiara il seguente costruttore privato:

```
/** Don't let anyone instantiate this class */
private System() {
}
```

Inoltre non è estendibile in quanto dichiarata `final`. Tutto ciò che esiste nella classe `System` è dichiarato statico.

12.2.1 Input e output

Abbiamo più volte utilizzato questa classe quando abbiamo usato l'istruzione per stampare `System.out.println()`. Senza saperlo, abbiamo utilizzato il metodo `println()` della classe `PrintStream` (del package `java.io`). Infatti il metodo è invocato sull'oggetto statico `out` della classe `System`, che è di tipo `PrintStream`. L'oggetto `out` rappresenta l'output di default del sistema che dipende dal sistema dove esegue l'applicativo.

Esiste anche l'oggetto `err (System.err)` che rappresenta l'error output di default del sistema, ovvero l'output di default dove vengono stampati gli errori. Anche esso è di tipo `PrintStream` e di solito coincide con l'output di cui sopra. Su un sistema Windows sia l'output di un programma che gli eventuali errori prodotti dal programma stesso, sono visualizzati sulla prompt DOS da dove si è eseguito l'applicativo. Esiste anche un oggetto statico `in (System.in)` che astrae il concetto di input di default del sistema. È di tipo `InputStream` (package `java.io` di cui parleremo più avanti) e solitamente individua la tastiera del computer. È possibile modificare il puntamento di queste variabili verso altre fonti di input o di output. Esistono infatti i metodi statici `setOut()`, `setErr()` e `setIn()`. Nel capitolo 18 relativo al package `java.io` (input-output) vedremo anche un esempio di utilizzo dell'oggetto `System.in` (leggeremo i caratteri digitati sulla tastiera) e come si può alterare l'indirizzamento di questi oggetti.

> **Negli esercizi che trovate on line, un grado di interazione con i nostri programmi è già stato introdotto dai primissimi capitoli.**

Dopo aver parlato delle variabili membro della classe `System`, diamo uno sguardo ai metodi più interessanti.

12.2.2 Le properties

`System` definisce diversi metodi interessanti, tra questi sicuramente ci sono quelli che gestiscono le properties come `setProperty(String key, String`

`value)` e `getProperty(String key)`. Per **system properties** intendiamo delle coppie chiave-valore che possono essere utilizzate dal programma. In particolare possono essere impostate all'interno e all'esterno del programma, e recuperate in qualsiasi parte dello stesso facilmente. Come abbiamo già affermato, la classe `System` astrae il sistema dove viene eseguita l'applicazione. Il sistema possiede determinate proprietà prestabilite. Per esempio:

```
  System.out.println(System.getProperty("java.version"));
```

restituirà la versione di Java che si sta utilizzando:

```
9.0.1
```

È possibile impostare nuove proprietà all'interno della classe `System`, come:

```
  System.setProperty("claudio.cognome", "De Sio Cesari");
```

Per poi recuperarle in un secondo momento all'interno di una qualsiasi classe che costituisce il programma, tramite il metodo `getProperty()`. È anche possibile impostare le properties nel momento in cui si esegue l'applicazione, mediante l'opzione `-D`. Per esempio se vogliamo eseguire un programma chiamato `SystemProperties` impostando la variabile `claudio.cognome`, possiamo scrivere:

```
java -Dclaudio.cognome="De Sio Cesari" SystemProperties
```

Si noti che tra la chiave della proprietà e l'opzione `-D` non ci devono essere spazi, e che siccome il valore è composto da più parole (`De Sio Cesari`), queste devono essere incluse tra virgolette per far comprendere che vanno considerate come un unico valore.

Il fatto di poter passare dall'esterno valori ad un programma può essere molto utile, per esempio allo scopo di configurarlo. Ancora più utile sarà poter leggere queste properties da un file, cosa che impareremo a fare quando affronteremo lo studio del capitolo 14 sul package `java.util`. Per eliminare una proprietà è possibile utilizzare il metodo `clearProperty()` in questo modo:

```
  System.clearProperty("claudio.cognome");
```

Questo metodo restituisce il valore precedentemente impostato per la proprietà, o `null` se questa non esisteva. Per un elenco completo di tutte le proprietà disponibili di default all'interno della classe `System`, è possibile usare il metodo `getProperties()`. Il seguente codice stampa tutte le proprietà del sistema con i relativi valori;

```
Properties properties = System.getProperties();
properties.list(System.out);
```

Con la prima riga abbiamo ricavato un oggetto `Properties` mediante il metodo `getProperties()` (che studieremo nel capitolo 14 dedicato al package `java.util`), e poi abbiamo chiamato su di esso il metodo `list()` che prende in input un oggetto `PrintStream` che sarà usato per stampare. Gli abbiamo passato un oggetto `System.out`, e quindi la lista delle proprietà sarà stampata sull'output di default.

12.2.3 Altri metodi di `System`

La classe `System` definisce il metodo `getenv()` che restituisce una mappa di tutte le variabili d'ambiente del sistema, come per esempio la variabile d'ambiente PATH, che abbiamo impostato in fase di installazione del JDK. È definito un suo overload che prende in input il nome della chiave della variabile d'ambiente da recuperare. Per esempio:

```
System.out.println("La variabile PATH vale: " +
System.getenv("PATH"));
```

stamperà la variabile d'ambiente di sistema PATH, dove dovreste trovare in testa il valore della cartella bin della cartella d'installazione del JDK.

Il metodo statico `exit(int code)` consente di bloccare istantaneamente l'esecuzione del programma. Il codice che viene specificato come parametro potrebbe servire al programmatore per capire perché si è interrotto il programma (un modo piuttosto rudimentale dopo aver visto la gestione delle eccezioni ed asserzioni). Segue un esempio di utilizzo di questo metodo:

```
if (continua == false) {
   System.err.println("Si è verificato un problema!");
   System.exit(0);
}
```

Il metodo `lineseparator()` restituisce la sequenza di caratteri di escape che rappresenta il separatore di riga, ovvero il carattere per andare da capo dipendente dal sistema operativo. Il metodo restituisce sempre lo stesso valore: il valore della proprietà di sistema `line.separator`. Questo per i sistemi Unix corrisponde a `\n`, mentre sui sistemi Windows a `\r\n`. Infatti i sistemi operativi interpretano i caratteri di escape in modo diverso. Per esempio `\n` viene detto "new line" (nuova linea) e viene interpretato da Windows come andare alla prossima linea, ma senza tornare all'inizio della riga. Quindi ha bisogno anche di `\r` che viene detto "carriage return" (ovvero "ritorno a capo") per tornare all'inizio della riga. Questo è importante quando si devono scrivere stringhe in un file, o in un flusso di rete perché su Windows `\n` non sarà sufficiente per essere interpretato come carattere per andare a capo. Per esempio aprendo con il Blocco Note di Windows un file che è stato scritto utilizzando il carattere `\n` in luogo del metodo `System.lineSeparator()`, scopriremo che il testo comprenderà delle anomalie.

Il metodo `arrayCopy()` consente di copiare gli elementi di un array in un altro. Questo perché essendo un array per definizione di dimensione fissa, c'era bisogno di un modo per simulare la ridimensionabilità degli array. Infatti alla base di numerose implementazioni del framework collection, ci sono array i cui elementi vengono copiati in altri array proprio mediante il metodo `arrayCopy()`. È un metodo che prende in input ben cinque parametri, il che lo rende difficile da utilizzare. Il primo parametro è di tipo `Object`, e rappresenta l'array di partenza (quello da cui si devono prendere gli elementi). Il secondo parametro è di tipo `int`, e rappresenta la posizione dell'array di partenza da cui si deve iniziare a copiare. Il terzo parametro è di tipo `Object` ancora e deve essere un array in cui copiare gli oggetti del primo array. Il quarto parametro è di tipo intero e sta ad indicare la posizione di partenza del secondo array in cui si deve iniziare a copiare. Infine il quinto parametro è ancora di tipo intero, e rappresenta il numero di elementi da copiare. Per esempio con il seguente codice:

```
int pari[] = {2, 4, 6, 8, 10} ;
int dispari[] = {1, 3, 5, 7, 9} ;
System.arraycopy(pari, 2, dispari, 1, 2);
System.out.print(Arrays.toString(dispari));
```

Abbiamo copiato dall'array `pari`, a partire dall'indice `2` (terzo elemento), nell'array `dispari` a partire dall'indice `1` (secondo elemento), due elementi (ovvero gli elementi `6` e `8`). Poi abbiamo stampato il contenuto del nuovo array `dispari` utilizzando il metodo statico `toString()` della classe d'utilità `java.util. Arrays` (che verrà trattato nel capitolo 17 relativo alle collections). L'output risultante è quindi il seguente:

```
[1, 6, 8, 7, 9]
```

> **I metodi statici `copyOf()` e `copyOfRange()` della classe `Arrays` possono fare più o meno le stesse operazioni di `arrayCopy()` (cfr. capitolo 17).**

Il metodo `currentTimeMillis()` è molto utilizzato. Esso restituisce il numero di millisecondi passati nel momento in cui lo si esegue a partire dal primo gennaio 1970. È un modo per ottenere un attimo nel tempo e calcolare il tempo che passa. Un classico esempio è quello in cui si calcola quanti millisecondi dura una certa operazione, che si può implementare in modo simile al seguente codice:

```
long tempoIniziale = System.currentTimeMillis();
//algoritmo non riportato
long tempoTotale = System.currentTimeMillis() -
tempoIniziale;
```

> **Esiste anche il metodo `nanoTime()` che può eseguire la stessa misurazione in nanosecondi.**

12.2.4 Garbage collection

La garbage collection è una delle caratteristiche più importanti di Java. Consiste in un processo della JVM che, in base ad un algoritmo, monitora la memoria ed interviene in certi frangenti per eliminare quella non più necessaria. Con Java 9, un nuovo

garbage collector denominato **G1GC** (dove con “GC” si intende “Garbage Collector”) è stato reso il garbage collector di default. G1GC non è una novità assoluta, in quanto è stato testato per diversi anni. Sin dalla versione 7 infatti, era possibile abilitare il G1GC come alternativa al Garbage Collector di default, denominato **ParallelGC.** Con Java 9 Oracle ha reputato i tempi maturi per l’aggiornamento. L’algoritmo di G1GC è molto differente rispetto a quello di ParallelGC. Invece di dividere la **memoria heap** (dove vengono allocati gli oggetti), in tre macroaree come fa ParallelGC, G1GC la suddivide in tante microaree di uguale grandezza. Ognuna di esse è monitorata e quando è il momento di intervenire, G1GC inizierà ad agire dall’area che ne ha più bisogno (da qui il nome G1GC ovvero “Garbage 1”, nel senso che la deallocazione parte sempre dall’area con maggiore priorità). Inoltre gli intervalli di intervento di G1GC saranno più brevi e frequenti, ed ottimizzeranno di volta in volta anche la frammentazione della memoria (ovvero ricompatteranno la memoria che avrà degli spazi vuoti lasciati dagli oggetti deallocati). Con ParallelGC gli interventi sono meno numerosi, più lunghi e intensi perché bisognerà ricompattare anche la memoria in diverse aree sparse. G1GC è oggi considerata più efficiente nella maggior parte delle applicazioni, mentre ParallelGC è appropriata per applicazioni dove lunghe pause non rappresentano un problema. Tuttavia è ancora possibile utilizzare ParallelGC (o anche altri garbage collector con algoritmi diversi come **SerialGC)** utilizzando l’opzione `-XX:+UseParallelGC` della JVM.

I metodi `runFinalization()` e `gc()` richiedono alla JVM rispettivamente la **finalizzazione degli oggetti inutilizzati** e eventualmente di liberarne la memoria. La finalizzazione consiste nel testare se esistono oggetti non più referenziati da qualche reference e quindi non utilizzabili. Questo viene compiuto mediante la chiamata al metodo `runFinalization()` definito nella classe `Runtime` (argomento del prossimo paragrafo). Nel caso in cui l’oggetto non sia più reputato utilizzabile dalla Java Virtual Machine, esso viene contrassegnato e il successivo passaggio della garbage collection (la chiamata al metodo `gc()`) dovrebbe deallocare la memoria allocata per l’oggetto in questione. Anche la chiamata al metodo `gc()` va a chiamare il metodo `gc()` della classe `Runtime`. Tuttavia l’utilizzo di questa coppia di metodi è sconsigliato perché il garbage collector stesso utilizza un meccanismo ottimizzato per gestire il momento giusto in cui chiamare questa coppia di

metodi. Per di più, questi due metodi fanno partire dei thread tramite il metodo `start()` che, come vedremo nel capitolo relativo ai thread, non garantisce l'esecuzione immediata del metodo stesso. Chiamare questi metodi significa solo chiedere alla JVM di attivare il processo del garbage collector, ma non è detto che la JVM eseguirà subito quanto richiesto.

12.3 La classe `Runtime`

La classe `Runtime` astrae il concetto di runtime (esecuzione) del programma. Non ha costruttori pubblici ed una sua istanza si ottiene chiamando il metodo factory `getRuntime()` (implementa il pattern **Singleton** spiegato nel capitolo 6). Tra le altre cose questa classe consente di eseguire comandi del sistema operativo direttamente da Java. Per esempio, il metodo `exec()` (di cui esistono più overload), utilizzato in questo modo:

```
Runtime r = Runtime.getRuntime();
try {
    r.exec("calc");
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

eseguirà la calcolatrice su un sistema Windows.

> **Bisogna tener conto che l'utilizzo del metodo `exec` potrebbe compromettere la portabilità delle nostre applicazioni. Infatti la classe `Runtime` è strettamente dipendente dall'interprete Java e quindi dal sistema operativo. Cercare di eseguire un programma come "calc" su Windows va bene, ma non su altri sistemi.**

Oltre ai metodi che `Runtime` ha in comune con `System`, esistono altri metodi nella classe `Runtime`. Per esempio il metodo `availableProcessors()` restituisce il numero di processori disponibili. I metodi `maxMemory()`, `freeMemory()` e `totalMemory()` restituiranno informazioni al runtime sullo

stato della memoria. In particolare `maxMemory()` restituisce la memoria massima che può utilizzare la JVM. Il metodo `freeMemory()` invece restituisce la memoria disponibile della JVM. Infine `totalMemory()` restituisce la memoria che sta utilizzando la JVM. L'unità di misura della memoria restituita è sempre da considerarsi il byte. Il seguente esempio è autoesplicativo:

```
public class RuntimeInfo {
    public static void main(String args[]) {
        Runtime runtime = Runtime.getRuntime();
        System.out.println("Il sistema dispone di "
                           + runtime.availableProcessors()
                           + " processori");
        System.out.println("Memoria totale allocata nella
JVM "
                           + runtime.totalMemory() + "
byte");
        System.out.println("Memoria libera nella JVM "
                           + runtime.freeMemory() + "
byte");
        System.out.println("Memoria massima allocabile dalla
JVM "
                           + runtime.maxMemory() + " byte");
    }
}
```

e produrrà un output simile al seguente:

```
Il sistema dispone di 8 processori
Memoria totale allocata nella JVM 268435456 byte
Memoria libera nella JVM 265329448 byte
Memoria massima allocabile dalla JVM 4273995776 byte
```

12.4 La classe `Class` e `Reflection`

La classe `Class` astrae il concetto di classe Java. Si usa per utilizzare una tecnica conosciuta come **reflection:** l'introspezione delle classi. Infatti la classe `Class` mette a disposizione metodi che si chiamano `getConstructor()`, `getMethods()`, `getFields()`, `getSuperClass()`, etc., i quali restituiscono oggetti di tipo `Costructor`, `Field`, `Method` e così via. Queste classi astraggono i concetti di costruttore, variabile e metodo. Queste

classi non si trovano all'interno del package `java.lang`, bensì si trovano all'interno del package `java.lang.reflect`. Ognuna di esse definisce metodi per ricavare informazioni specifiche. Per esempio la classe `Field` dichiara il metodo `getModifiers()` che restituisce i modificatori della variabile d'istanza.

12.4.1 Istanziare un oggetto `Class`

Si può istanziare una classe `Class`, in tre modi diversi. Un primo modo consiste nello sfruttare il metodo statico `forName(String name)`. A partire da una stringa si può istanziare un oggetto `Class` in questo modo:

```
try {
    Class stringa = Class.forName("java.lang.String");
} catch (ClassNotFoundException exc) {
    //...
}
```

Un secondo modo consiste nell'ottenere un'istanza invocando il metodo `getClass()` su un oggetto:

```
String a = "MiaStringa";
Class stringa = a.getClass();
```

> **Si noti che il metodo `getClass()` è un altro metodo definito nella classe `Object`.**

Il terzo modo consiste nell'utilizzare un cosiddetto **class literal** a partire da un certo tipo:

```
Class stringa = java.lang.String.class;
```

un class literal definisce appunto un oggetto di tipo `Class`, a partire da un tipo, a cui si aggiunge il suffisso .class. Riassumendo, possiamo ottenere un'istanza di una classe `Class` a partire da una stringa usando il metodo `forName()`, a partire da un oggetto invocando il metodo `getClass()` ereditato dalla classe `Object`, o da un tipo utilizzando un class literal.

Nella reflection la classica istanza della classe `Class` si ottiene tramite la stringa del **nome completo** della classe (in inglese **fully qualified name)** ovvero il nome comprensivo di package. Spesso questa tecnica può diventare molto utile, ma bisogna fare attenzione quando si ha a che fare con le stringhe. Il compilatore non ne controllerà la correttezza, quindi un piccolo errore sintattico creerà un baco in esecuzione.

12.4.2 `Class` è un tipo generico

La classe `java.lang.Class` è un tipo generico dichiarato nel seguente modo:

```
public final class Class<T>
```

Il parametro `T` rappresenta la classe che sarà usata dall'istanza dell'oggetto `Class` istanziato. Questo ci consentirà di evitare cast, e di sapere a priori cosa contiene un oggetto `Class`. Un semplice esempio di utilizzo di `Class` potrebbe essere il seguente:

```
Class<Punto> puntoClass = Punto.class;
Punto punto =
puntoClass.getDeclaredConstructor().newInstance();
```

dove, dopo aver ottenuto un oggetto `puntoClass` della classe `Class`, istanziamo un oggetto `punto` da esso chiamando il metodo `newInstance()` sull'oggetto `Constructor` ritornato dal metodo `getDeclaredConstructor()`. Quest'ultimo prende in input un varargs di oggetti `Class` che rappresentano i tipi dei parametri. Siccome non abbiamo passato nulla, allora ritorna il costruttore senza parametri. Si noti che non è stato necessario nessun cast. La parametrizzazione della classe `Class` permette di scrivere metodi complessi come il seguente:

```
public class GenericClassInstanceDemo {
     public static <T> T createObjectFromClass(Class<T>
type)
       throws Exception {
         T object =
type.getDeclaredConstructor().newInstance();
         return object;
     }
}
```

che crea un oggetto dalla classe passata in input. Nell'esempio di utilizzo che segue:

```
MiaClasse miaClasse =
  GenericClassInstanceDemo.createObjectFromClass(MiaClasse.clas
```

nessun cast è stato eseguito.

> **Si noti che in questo caso abbiamo parametrizzato anche il metodo `createObjectFromClass()` con il parametro `<T>`.**

12.4.3 Un esempio di reflection

Il seguente esempio permette di stampare le firme di tutti i metodi della classe passata da riga di comando (se viene eseguito senza specificare parametri esplora la classe `Object`):

```
import java.lang.reflect.*;

public class TestClassReflection {
    public static void main(String args[]) throws Exception
{
    String className = "java.lang.Object";
    if (args.length > 0) {
        className = args[0];
    }
    Class<?> objectClass = Class.forName(className);
    Method[] methods = objectClass.getMethods();
    for (int i = 0; i < methods.length; i++) {
        String name = methods[i].getName();
        Parameter[] classParameters =
methods[i].getParameters();
        String stringClassParameters = "";
        int length = classParameters.length;
        for (int j = 0; j < length; ++j) {
            stringClassParameters +=
              classParameters[j].getType().getName() + " " +
              classParameters[j].getName();
            // aggiungiamo una virgola per separare i
parametri
            // ma solo se non è l'ultimo parametro
```

```
            if (j != (length-1)) {
                stringClassParameters += ", ";
            }
        }
        String methodReturnType =
methods[i].getReturnType().getName();
        String methodString = methodReturnType + " " +
          name + " (" + stringClassParameters + ")";
        System.out.println(methodString);
     }
   }
}
```

Questa classe dichiara la variabile locale `className` di tipo stringa, assegnandogli il fully qualified name `java.lang.Object`. Il valore della variabile sarà sovrascritto se verrà passato un parametro al metodo `main()` da riga di comando. Per esempio potremmo eseguire questa classe in questo modo:

```
java TestClassReflection java.lang.String
```

così nella variabile `className` sarà immagazzinato il valore `java.lang.String`. Poi abbiamo istanziato la variabile `objectClass` di tipo `Class<?>`. Abbiamo usato una wildcard perché non sappiamo a priori questa `Class` a che tipo si riferirà. Quindi tramite il metodo `getMethods()` della classe `Class` ci siamo ricavati un array di oggetti `Method` che chiamiamo `methods`. Con un ciclo su questo array creiamo un algoritmo che stampa le firme dei metodi. All'interno del ciclo, immagazziniamo il nome del metodo corrente nella variabile `name`. Poi recuperiamo gli oggetti `classParameters` con il metodo `getParameters()`. Prima di entrare nel ciclo sui parametri del metodo dichiariamo la stringa `stringClassParameters`, in cui immagazzineremo la stringa che stamperà in maniera leggibile i parametri del metodo corrente. Entrati nel secondo ciclo, per ogni parametro aggiungiamo alla stringa `stringClassParameters` il tipo di parametro, il nome del parametro e una virgola (tranne nell'ultima iterazione). Nella ultima parte del metodo, usciti dal ciclo dei parametri, andiamo a formattare e stampare la stringa del metodo corrente (variabile `methodString`), aggiungendo il tipo di ritorno, il nome e la lista dei parametri tra parentesi tonde. Per esempio passando a questo metodo la classe `Object`, otterremo il seguente output:

```
java TestClassReflection Punto

void wait (long arg0, int arg1)
void wait ()
void wait (long arg0)
boolean equals (java.lang.Object arg0)
java.lang.String toString ()
int hashCode ()
java.lang.Class getClass ()
void notify ()
void notifyAll ()
```

12.4.4 Nomi dei parametri dei metodi

Nell'esempio precedente possiamo notare che è stata utilizzata la classe `Parameter`, che è stata introdotta con Java 8 e che astrae il concetto di parametro di un metodo. È possibile quindi tramite reflection recuperare anche il nome del parametro del metodo. È importante notare però, che di default nel bytecode non sono preservati i nomi dei parametri dei metodi. Se provassimo quindi ad eseguire questo programma passandogli una classe compilata da noi, i nomi dei parametri dei metodi non potranno essere recuperati, e saranno riportati come `args0`, `args1`, `args2` e così via. Per esempio se eseguiamo l'esempio precedente passando come parametro la classe `Punto`:

```
java TestClassReflection Punto
```

allora otterremo il seguente output:

```
boolean equals (java.lang.Object arg0)
int hashCode ()
void setX (int arg0)
void setY (int arg0)
int getX ()
int getY ()
void wait (long arg0, int arg1)
void wait ()
void wait (long arg0)
java.lang.String toString ()
java.lang.Class getClass ()
void notify ()
void notifyAll ()
```

Per permettere al compilatore di conservare anche i nomi dei parametri dei metodi, bisogna utilizzare il flag `-parameters` quando si compila la classe da cui leggere i parametri. Per esempio, compilando la classe `Punto` nel seguente modo:

```
javac -parameters NomeClasse.java
```

ed eseguendo l'esempio precedente passando come parametro la classe `Punto`:

```
java TestClassReflection Punto
```

allora otterremo il seguente output:

```
java TestClassReflection Punto
boolean equals (java.lang.Object altroOggetto)
int hashCode ()
void setY (int y)
void setX (int x)
int getX ()
int getY ()
void wait (long arg0, int arg1)
void wait ()
void wait (long arg0)
java.lang.String toString ()
java.lang.Class getClass ()
void notify ()
void notifyAll ()
```

Possiamo notare che i nomi dei parametri della classe `Punto` (`x` e `y`) ora sono quelli originali.

12.5 La classe `String`

La classe `String` è stata introdotta sin dal capitolo 3, ed è già stata utilizzata più volte in questo testo. Dato che si tratta probabilmente della classe più utilizzata nella programmazione Java, i creatori di Java decisero che sarebbe stata l'unica classe istanziabile come fosse un tipo di dato primitivo. Infatti quando scriviamo:

```
  String stringa = "Questa è una stringa";
```

istanziamo una stringa con un **literal** (cfr. paragrafo 3.2.4). Anche se abbiamo utilizzato una sintassi semplificata senza utilizzare un costruttore e l'operatore `new`, abbiamo comunque istanziato un oggetto. Ma `String` è una classe speciale anche per altre ragioni. Per esempio le istanze di `String` sono **immutabili**, ovvero non possono essere modificate una volta create. Infatti i metodi di questa classe non modificano mai il loro contenuto, al massimo restituiscono un'altra stringa. Inoltre, le istanze che vengono create con un literal come se `String` fosse un tipo di dato primitivo, vengono poste in uno speciale **pool di stringhe** (un insieme speciale) dalla Java Virtual Machine e riutilizzate, al fine di migliorare le prestazioni.

12.5.1 Gestione in memoria

Con il termine **pool di stringhe** ci riferiamo ad una particolare soluzione architetturale (un pattern) che consente di riutilizzare le stringhe già istanziate. Un pool di stringhe è un oggetto in cui la JVM conserva le istanze di `String` che sono create senza la parola chiave `new`, come se `String` fosse un tipo di dato primitivo. Nel caso, durante l'esecuzione del programma sia istanziata allo stesso modo un'altra stringa, e questa risulti già presente nel pool di stringhe, essa non sarà istanziata, ma verrà riciclata l'istanza presente nel pool. Questo tipo di approccio è sicuro perché gli oggetti stringa sono **immutabili**, ovvero una stringa non si può modificare. Come vedremo nel prossimo paragrafo infatti, i metodi di `String` che sembrano modificare una stringa in realtà ne restituiscono una nuova.

Proviamo a fare un esempio confrontando le stringhe con l'operatore ==. Come abbiamo visto nel capitolo 3, quando esso è applicato a due operandi complessi, restituisce `true`, se e solo se i due oggetti coincidono (ovvero i due reference puntano alla stessa area di memoria). Infatti l'operatore == confronta i valori dei due reference, e questi valori rappresentano l'indirizzo in memoria a cui puntano. Istanziamo la classe `String` con un literal con la sintassi semplificata:

```
String linguaggio = "Java";
```

Le istanze ottenute in questo modo, sono trattate diversamente da quelle istanziate con la sintassi tradizionale valida per tutte le altre classi:

```
String linguaggio = new String("Java");
```

Infatti, le istanze create con la sintassi semplificata, sono poste in un pool di stringhe dalla JVM e riutilizzate. Verifichiamolo con il seguente frammento di codice:

```
String a = "Java";
String b = "Java";
String c = new String("Java");
System.out.println(a==b);
System.out.println(b==c);
```

il codice precedente produrrà il seguente output:

```
true
false
```

Questo perché gli oggetti `a` e `b` puntano esattamente allo stesso oggetto. Mentre nel caso di `c`, avendo utilizzato la parola chiave `new`, sarà creato un nuovo oggetto, anche se con lo stesso contenuto dei precedenti. Ricordiamo che il giusto modo per confrontare due stringhe (come per tutti gli oggetti) rimane l'utilizzo del metodo `equals()`. Infatti, il seguente frammento di codice:

```
System.out.println(a.equals(b));
System.out.println(b.equals(c));
```

produrrà il seguente output:

```
true
true
```

Nel caso si voglia confrontare un reference a stringa e un valore literal di stringa, si corre il rischio di andare incontro ad una `NullPointerException`. Per esempio se fosse definito il seguente `metodo`:

```
public void esegui(String comando) {
```

```
      if (comando.equals("AVANTI")) {
         //...
      } else if //...
  }
```

questo risulterebbe potenzialmente pericoloso. Infatti se a questo metodo fosse passata una stringa `null`, allora andremo incontro ad una `NullPointerException`. Invece se lo riscrivessimo così:

```
  public void esegui(String comando) {
      if ("AVANTI".equals(comando)) {
         //...
      } else if //...
  }
```

non correremmo nessun rischio. Si consiglia quindi preferire come oggetto su cui chiamare il metodo `equals()` sempre una stringa literal rispetto ad un reference.

12.5.2 Dichiarazione della classe `String`

Nella figura 12.2 è mostrata la pagina della documentazione standard relativa alla dichiarazione della classe `String`.

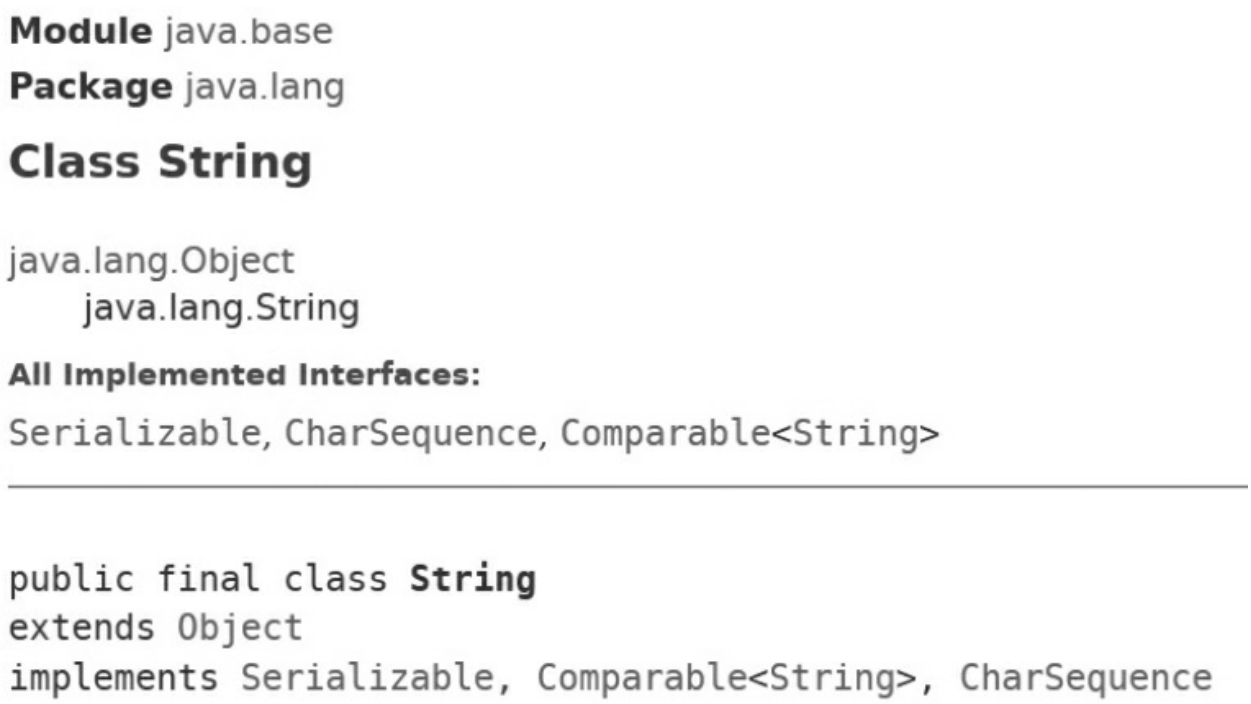


Figura 12.2 - Documentazione della classe String.

Osserviamo che questa classe implementa l'interfaccia `CharSequence` appartenente allo stesso package `java.lang`. Quando incontreremo metodi che dichiarano come parametri `CharSequence`, potremo quindi utilizzare le stringhe come parametri polimorfi.

> **Tale interfaccia è implementata anche da altre classi, tra le quali la più famosa è `Stringbuilder`, che incontreremo presto durante lo studio di questo testo.**

La classe `String` implementa inoltre anche l'interfaccia `Comparable`, sempre appartenente allo stesso package `java.lang`. Questa è progettata per permettere di ordinare oggetti all'interno di una collection. La classe `String` implementa il metodo ereditato `compareTo()` (spiegato nelle prossime righe) che definisce il proprio ordinamento naturale. L'interfaccia `Comparable`, sarà approfondita nel paragrafo 12.8.1 di questo capitolo. `String` implementa anche l'interfaccia `Serializable` appartenente al package `java.io`. Questa interfaccia non ha metodi, ed è implementata dalle classi che devono poter essere serializzate, ovvero salvate in un certo momento e in un certo stato. Di questo argomento parleremo nel capitolo 18 relativo all'Input-Output. Infine la classe `String` per ragioni di sicurezza è dichiarata `final`, e quindi non si può estendere.

12.5.3 I metodi di `String`

Di seguito sono elencati i metodi più importanti della classe `String`, con descrizione ed esempi.

12.5.3.1 `char charAt(int index)`

Restituisce il carattere all'indice specificato (indice iniziale `0`). Per esempio:

```
System.out.println("Non è logico, ma è spesso
vero.".charAt(2));
```

stampa:

```
n
```

12.5.3.2 `String concat(String other)`

Restituisce una nuova stringa che concatena la vecchia con la nuova `(other)`. Il seguente codice:

```
 String stringa1 = "Mc";
 String stringa2 = "Guffin";
 String stringa3 = stringa1.concat(stringa2);
 System.out.println("stringa1 = " + stringa1);
 System.out.println("stringa2 = " + stringa2);
 System.out.println("stringa3 = " + stringa3);
```

stamperà:

```
stringa1 = Mc
stringa2 = Guffin
stringa3 = McGuffin
```

12.5.3.3 `int compareTo(String other)`

Esegue una comparazione lessicale: ritorna un intero `< 0` se la stringa corrente è minore della stringa `other`, un intero `= 0` se le due stringhe sono identiche e un intero `> 0` se la stringa corrente è maggiore di `other`. È l'implementazione del metodo ereditato dall'interfaccia funzionale `Comparable` (cfr, paragrafo 12.8.1) implementata da `String`. Quindi l'output dello snippet:

```
 System.out.println("A".compareTo("B"));
 System.out.println("C".compareTo("B"));
 System.out.println("D".compareTo("D"));
```

stamperà:

```

-1
1
0
```

Dobbiamo interpretare i risultati così: `A` minore di `B`, `B` minore di `C`, e naturalmente `D` è uguale a `D`.

12.5.3.4 `boolean endsWith(String suffix)`

Restituisce `true` se e solo se la stringa corrente termina con `suffix`. Per esempio:

```
String nome = "Viola";
if (nome.endsWith("a")) {
    System.out.println(nome + " termina con la \"a\"");
}
```

stamperà:

```
Viola termina con la "a"
```

12.5.3.5 `boolean equals(Object ob)`

È un override dell'omonimo metodo di `Object` e restituisce `true` se e solo se il contenuto della stringa corrente è uguale alla stringa passata come parametro. Per esempio il seguente codice ha valore `false`:

```
new String("sapore").equals("Sapore"))
```

12.5.3.6 `boolean equalsIgnoreCase(String s)`

È un metodo equivalente ad `equals()` che ignora la differenza tra lettere maiuscole e minuscole. Per esempio il seguente codice ha valore `true`:

```
new String("battista").equalsIgnoreCase("Battista"))
```

12.5.3.7 `Static String format(String format, Object... args)`

È un metodo statico che permette di formattare una stringa parametrizzandola con una sintassi specifica. Per esempio se all'interno della stringa utilizziamo `%s`, questo potrà essere sostituito con una stringa passata come parametro nel varargs, mentre con `%b` può essere sostituito con un booleano. Ad esempio:

```
System.out.println(String.format("Stringa = %s, booleano =
%b", "a", 1==1));
```

stamperà:

```
Stringa = a, booleano = true
```

I dettagli di questo metodo saranno specificati nel paragrafo 13.1.5. Ne esiste anche una versione per gestire l'internazionalizzazione (cfr. paragrafo 13.1.2).

12.5.3.8 `int indexOf(int ch)`

Restituisce l'indice del carattere specificato. Si noti che in realtà potremmo passare anche un intero come parametro. Questo perché così sarà possibile anche passare la rappresentazione Unicode del carattere da cercare (per esempio `0XABCD`). Il metodo restituisce `-1` nel caso non sia trovato il carattere richiesto. Se esistessero più occorrenze nella stringa del carattere richiesto, verrebbe restituito l'indice della prima occorrenza. Il seguente statement restituisce il valore `1`:

```
"Ligeia".indexOf("i"));
```

12.5.3.9 `int indexOf(int ch, int fromIndex)`

È equivalente al metodo precedente ma la stringa viene presa in considerazione dall'indice specificato dalla variabile `fromIndex` in poi. Di questi due ultimi metodi esistono versioni con `String` al posto di `int`. Il seguente statement restituisce il valore `4`:

```
"Ligeia".indexOf("i", 2));
```

12.5.3.10 `String intern()`

Prova a recuperare l'oggetto `String` dal pool di stringhe, utilizzando il confronto che fornisce il metodo `equals()`. Nel caso nel pool non esista la stringa desiderata, questa viene aggiunta e viene restituito un reference ad essa. Si può applicare questo metodo alle stringhe istanziate con la sintassi che fa uso di costruttore, per far sì che siano utilizzate come le stringhe

istanziate con la sintassi semplificata (ovvero che siano memorizzate nel pool per poter essere riutilizzate).

12.5.3.11 `static String join(CharSequence delimiter, CharSequence... elements)`

Questo metodo statico introdotto in Java 8, ritorna una nuova stringa composta da copie degli elementi del varargs `elements` specificato (ricordiamo che `CharSequence` è implementato da `String`) separate dal delimitatore specificato. Per esempio:

```
String message = String.join("\", "C:", "Esempi", "Java");
System.out.println(message);
```

stamperà:

```
C:\Esempi\Java
```

12.5.3.12 `static join(CharSequence delimiter, Iterable<? extends CharSequence> elements)`

È un overload del metodo precedente ed usa un oggetto che implementa `Iterable` (come un `ArrayList`) di `CharSequence` (per esempio stringhe) in luogo di un varargs come ultimo parametro.

12.5.3.13 `int lastIndexOf(int ch)`

È come `indexOf()` ma viene restituito l'indice dell'ultima occorrenza trovata del carattere specificato. Il seguente codice restituisce il valore `6`:

```
"Rosalia".lastIndexOf("a");
```

12.5.3.14 `int length()`

Restituisce il numero di caratteri di cui è costituita la stringa corrente. Quindi:

```
"Claudio".length();
```

restituirà `7`.

12.5.3.15 `String replace(char oldChar, char newChar)`

Restituisce una nuova stringa, dove tutte le occorrenze di `oldChar` sono rimpiazzate con `newChar`.

12.5.3.16 `String replace(CharSequence target, CharSequence replacement)`

Funziona come il metodo precedente ma utilizza `CharSequence` come parametri (ovvero stringhe).

12.5.3.17 `String replaceAll(String regex, String replacement)`

Restituisce una nuova stringa partendo dalla stringa corrente e sostituendo ogni occorrenza della regular expression `regex` specificata con la stringa `replacement`. Le regular expression (espressioni regolari) saranno spiegate nel capitolo 14 relativo alla libreria `java.util`.

12.5.3.18 `String replaceFirst(String regex, String replacement)`

Sostituisce la prima occorrenza di questa stringa che coincide con la regular expression `regex` specificata con la stringa `replacement`. Per esempio:

```
String newString = "aomame".replaceFirst("a", "A");
System.out.println(newString);
```

stamperà:

```
Aomame
```

12.5.3.19 `String[] split(String regex)`

Sfruttando la specificata regular expression `regex`, divide la stringa in un array di stringhe. Per esempio:

```
String deepPurple = "Gillan-Glover-Paice-Blackmore-Lord";
String[] components = deepPurple.split("-");
```

L'array `components` conterrà i seguenti valori: `"Gillan"`, `"Glover"`, `"Paice"`, `"Blackmore"`, `"Lord"`.

12.5.3.20 `boolean startsWith(String prefix)`

Restituisce `true` se e solo se la stringa corrente inizia con la stringa `prefix`.

12.5.3.21 `boolean startsWith(String prefix, int fromIndex)`

È equivalente al metodo precedente ma la stringa viene presa in considerazione dall'indice specificato dalla variabile `fromIndex` in poi.

12.5.3.22 `String substring(int startIndex)`

Restituisce una sottostringa della stringa corrente, composta dai caratteri che partono dall'indice `startIndex` alla fine.

12.5.3.23 `String substring(int startIndex, int endIndex)`

Restituisce una sottostringa della stringa corrente, composta dai caratteri che partono dall'indice `startIndex` fino all'indice `endIndex - 1`. Quindi l'output di questo snippet:

```
 String cantoPrimo = "Nel mezzo del cammin di nostra vita mi
 "
   + "ritrovai per una selva oscura, ché la diritta via era
 smarrita";
 System.out.println(cantoPrimo.startsWith("mez"));
 System.out.println(cantoPrimo.startsWith("mez", 4));
```

sarà:

```
false
true
```

12.5.3.24 `String toLowerCase()`

Restituisce una nuova stringa equivalente a quella corrente ma con tutti i

caratteri minuscoli.

12.5.3.25 `String toString()`

Restituisce la stringa corrente (!)... comunque è ereditato dalla classe `Object`.

12.5.3.26 `String toUpperCase()`

Restituisce una nuova stringa equivalente a quella corrente ma con tutti i caratteri maiuscoli.

12.5.3.27 `String trim()`

Restituisce una nuova stringa eliminando i caratteri spazio (`'\n'` e `'\r'`) che eventualmente si trovano all'inizio o alla fine della stringa corrente.

```
String stringToTrim = "     Emanuele     ";
final String STAR ="*";
System.out.println(STAR + stringToTrim.trim() + STAR);
```

stamperà:

```
*Emanuele*
```

12.5.3.28 `static String valueOf(Object obj)`

Ritorna il valore della chiamata di `toString()` sull'oggetto passato in input, oppure `null` nel caso l'oggetto in input sia `null`. Rispetto al chiamare direttamente il metodo `toString()` sull'oggetto, questo metodo ci mette al sicuro da eventuali `NullPointerException`, che potrebbero essere generate dal chiamare il metodo `toString()` su un oggetto `null`. Di questo metodo sono implementati diversi overload per altri tipi di dati primitivi (cfr. documentazione ufficiale).

> **Si noti che la classe di utilità `Objects` del package `java.util`, fornisce un metodo statico del tutto simile a `valueOf()` di `String`, ma più completo. Infatti prende in input due parametri: il primo è l'oggetto su cui chiamare il**

metodo `toString()`**, il secondo la stringa da ritornare nel caso il primo parametro sia** `null`.

Esistono tanti altri metodi ed overload di quelli elencati, che possono essere consultati direttamente tramite la documentazione ufficiale.

12.5.4 Compact Strings

La classe `String` è indubbiamente la classe più utilizzata nella programmazione Java, ed è quindi necessario porsi il problema di quanto gli oggetti di questa classe siano efficienti. Sino a Java 8, all'interno della classe era utilizzato un array di caratteri per memorizzare i caratteri della stringa. Era possibile verificarlo andando a guardare i file sorgenti della libreria Java, che sono contenuti in un file chiamato src.zip posizionato nella cartella di installazione del JDK. All'interno di esso navigando nella cartella java e poi nella cartella lang potevamo trovare il sorgente della classe String.java (infatti la classe `String` appartiene al package `java.lang`):

```
public final class String
    implements java.io.Serializable, Comparable<String>,
CharSequence {
    /** The value is used for character storage. */
    private final char value[];
    //omesso il resto del codice
```

Sino a Java 8 quindi, l'array di caratteri `value` implicava che per ogni carattere di una stringa era necessario utilizzare 16 bit. Il problema però, era che per immagazzinare un carattere di uso comune, in realtà bastano solo 8 bit. Quindi per ottenere un significativo risparmio della memoria dei programmi Java, in Java 9 l'implementazione della classe `String` è stata rivista per essere supportata da un array di `byte` invece che da un array di `char`.

Nel JDK 9 il file src.zip è stato spostato nella cartella lib**, e i package sono stati inclusi nelle cartelle che rappresentano i**

moduli. Quindi il sorgente String.java **si trova ora sotto le cartelle** java.base/java/lang. **Infatti** `java.base` **è il nome del modulo che contiene il package** `java.lang`.

È sempre possibile utilizzare caratteri meno comuni, perché internamente alla classe `String`, è stato implementato un meccanismo basato sulla variabile `coder` che si occupa di capire in quanti byte si deve memorizzare ogni carattere. Questo nuovo modo di immagazzinare stringhe è detto **compact strings**, e in Java 9 è il metodo utilizzato di default dalla JVM. In questo modo la memoria viene ottimizzata ma non è detto che le performance dei programmi Java saranno migliori. È possibile comunque disabilitare le compact strings utilizzando l'opzione `-XX:-CompactStrings`. Nel seguente programma:

```
public class CompactStringsDemo {
    public static void main(String[] args) {
        Runtime runtime = Runtime.getRuntime();
        long dimensioneInizialeMemoria =
runtime.totalMemory() -
                                        runtime.freeMemory();

        long tempoIniziale = System.currentTimeMillis();
        long limite = 100_000;
        String s = "";
        for (int i = 0; i < limite; i++) {
            s += limite;
         }
         long tempoTotale = System.currentTimeMillis() -
tempoIniziale;
         System.out.println("Create " + limite + " stringhe
in "
                            + tempoTotale + "
millisecondi");
         long dimensioneFinaleMemoria =
runtime.totalMemory() -
                                       runtime.freeMemory();

         long memoriaUtilizzata = dimensioneFinaleMemoria -
                                  dimensioneInizialeMemoria;
         System.out.println("Memoria utilizzata = " +
memoriaUtilizzata
```

```
                                + " byte");
        }
    }
```

è eseguita l'istanza di 100.000 stringhe (che contengono proprio i primi 100.000 numeri) che vengono concatenate. Inoltre sono calcolati i millisecondi che occorrono per creare queste istanze e concatenarle, ed è anche calcolata la memoria utilizzata della JVM all'inizio e alla fine del programma. Eseguendo l'applicazione disabilitando le compact strings otterremo il seguente output:

```
java -XX:-CompactStrings CompactStringsDemo
Create 100000 stringhe in 10972 millisecondi
Memoria utilizzata = 296515472 byte
```

Eseguendo l'applicazione senza disabilitare le compact strings invece:

```
java CompactStringsDemo
Create 100000 stringhe in 5199 millisecondi
Memoria utilizzata = 150139312 byte
```

In questo caso possiamo notare un incremento delle prestazioni e della gestione della memoria vicino al 50%. Bisogna però essere consapevoli che si tratta solo di un esempio, e quindi questi numeri non saranno sempre a favore delle compact strings. Esistono anche situazioni nelle quali disabilitando le compact strings le performance miglioreranno.

12.6 Le classi wrapper

Sono dette **classi wrapper** (in italiano "involucro") le classi che fanno da contenitore ad un tipo di dato primitivo, astraendo proprio il concetto di tipo primitivo. In Java esistono le classi `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`, `Character`, ognuna delle quali può contenere il relativo tipo primitivo.

> **Le classi wrapper** `Integer`, `Byte`, `Short`, `Long`, `Character`, `Double` **e** `Float` **(l'unica mancante è la classe** `Boolean`**) sono tutte sottoclassi di** `Number`.

Come la classe `String`, anche le classi wrapper sono **immutabili.** Questo implica che un oggetto istanziato da una classe wrapper non potrà mai cambiare il suo stato interno tramite un suo metodo, infatti non esistono metodi setter. Quindi un oggetto `Boolean` istanziato così:

```
Boolean b = new Boolean(true);
```

non conterrà mai il valore `false`. Le classi wrapper sono utili ed indispensabili soprattutto nei casi in cui dobbiamo utilizzare un tipo di dato primitivo laddove è richiesto un oggetto. Per esempio, supponiamo di voler dichiarare una lista generica di interi. Non è possibile dichiarare la lista in questo modo:

```
ArrayList<int> list = new ArrayList<>();
```

Perché i generici richiedono tipi parametro solo complessi e non primitivi. Ma è possibile dichiarare:

```
ArrayList<Integer> list = new ArrayList<>();
```

che soddisfa i nostri requisiti grazie alle regole dell'autoboxing-autounboxing.

12.6.1 Autoboxing-Autounboxing

Un tipico esempio in cui è necessario utilizzare le classi wrapper riguarda quindi l'utilizzo delle collections. Infatti esse accettano come elementi solo oggetti e non tipi di dati primitivi. Tuttavia grazie all'**autoboxing-autounboxing** è possibile aggiungere tipi primitivi direttamente alle collections. Questa caratteristica consente di fatto di equiparare i tipi primitivi ai relativi tipi wrapper. Quindi sarà possibile tranquillamente aggiungere un tipo primitivo ad una `Collection`, come nel seguente esempio:

```
List<Float> list = new ArrayList<>();
list.add(5.3F);
```

Il compilatore di Java convertirà il codice precedente in modo tale che il valore primitivo `5.3F` sia dapprima inscatolato automaticamente (da qui il termine autoboxing) nel relativo tipo wrapper (in questo caso un `Float`).

Ovvero, è come se il codice dell'esempio precedente fosse trasformato dal compilatore nel seguente codice:

```
ArrayList<Float> list = new ArrayList<>();
list.add(new Float(5.3F));
```

Sarà anche possibile estrarre direttamente il tipo primitivo dalla `Collection` senza alcun passaggio intermedio né casting, grazie all'autounboxing:

```
float primitiveFloat = list.get(0);
```

Dal momento che i tipi primitivi ed i relativi tipi wrapper sono di fatto equiparati, sarà possibile eseguire operazioni aritmetiche incrociate come nei seguenti esempi:

```
Integer i = new Integer(22);          //i vale 22
int j =i++;                           //j vale 22, i vale 23
Integer k = (new Integer (10)+ j);     //k vale 32
int t = k + j + i;                    //t vale 32 + 22 + 23
= 77
```

Il valore di `t` sarà `77` (tenere presente le regole di priorità degli operatori, cfr. capitolo 3). Il seguente codice:

```
int i = 1;
Integer integer = new Integer(2);
int somma = i + integer;
```

compila correttamente. Lo stesso dicasi per:

```
Integer somma = i + integer;
```

Seguono altri esempi di codice valido:

```
Integer i = 0;
Double d = 2.2;
char c = new Character('c');
```

In definitiva, un `int` ed un `Integer` sono equivalenti e così tutti i tipi primitivi con i rispettivi wrapper.

La seguente istruzione invece è illegale:

```
Double d = 2;
```

Infatti il valore `2` è di tipo `int`. L'autoboxing richiede che ci sia una coincidenza perfetta tra il tipo primitivo e il relativo tipo wrapper. Non importa che un `int` possa essere contenuto in un `double`. Il problema è facilmente risolvibile con il seguente cast (cfr. capitolo 3):

```
Double d = 2D;
```

Questo favorisce la risoluzione di alcune criticità come l'inserimento di dati primitivi nelle collections. Prima di Java 5, per esempio, il codice:

```
ArrayList list = new ArrayList();
list.add(l);
list.add(3.14F);
list.add('c');
```

avrebbe provocato errori in compilazione. Infatti, non era possibile aggiungere tipi primitivi laddove ci si aspetta un tipo complesso. L'equivalente codice prima di Java 5 doveva essere il seguente:

```
ArrayList<Number> list = new ArrayList<>();
list.add(new Integer(1));
list.add(new Float(3.14F));
list.add(new Character('c'));
```

Per poi dover recuperare i dati primitivi successivamente, mediante codice di questo tipo:

```
Integer i = (Integer) list.elementAt(0);
Float p = (Float) list.elementAt(1);
Character c = (Character) list.elementAt(2);
int intero = i.intValue();
float floating = b.floatValue();
character c = c.charValue();
```

Oggi invece è possibile scrivere direttamente:

```
ArrayList<Number> list = new ArrayList<>();
list.add(1);
list.add(false);
list.add('c');
int intero = (Integer) list.elementAt(0);
boolean booleano = (Boolean) list.elementAt(1);
character c = (Character) list.elementAt(2);
```

e questo non può che semplificare la vita al programmatore.

> **Il termine "boxing" equivale al termine italiano "inscatolare". L'inscatolamento dei tipi primitivi nei relativi tipi wrapper è automatico (ci pensa il compilatore). Ecco perché si parla di autoboxing. L'autounboxing invece è il processo inverso. Si tratta quindi di una doppia funzionalità. Il compilatore non fa altro che inscatolare i tipi primitivi nei relativi tipi wrapper, o estrarre tipi primitivi dai tipi wrapper, quando ce n'è bisogno. L'autoboxing e l'autounboxing sono comodità forniteci dal compilatore che svolge il lavoro per noi.**

12.6.1.1 Gestione in memoria dei tipi primitivi

Le specifiche di Java asseriscono che alcuni tipi di dati primitivi sono sempre inscatolati nelle stesse istanze wrapper immutabili. Tali istanze sono poi poste in uno speciale **pool** (del tutto simile a quello discusso per le stringhe) dalla Virtual Machine e riusate, perché ritenute di frequente utilizzo. Tutto questo col fine di migliorare le performance. Godono di questa caratteristica:

1. tutti i tipi `byte`;
2. i tipi `short` e `int` con valori compresi nell'ordine dei `byte` (da `-128` a `127`);
3. i tipi `char` con range compreso da `\u0000` a `\u007F` (cioè da `0` a `127`);
4. i tipi `boolean`.

Vedremo presto che, benché questa osservazione sembri fine a sé stessa, ha un'importante conseguenza di cui i nostri programmi devono tener conto.

12.6.1.2 Assegnazione di un valore `null` al tipo wrapper

Non sempre l'autoboxing-autounboxing è così semplice. Per esempio è sempre possibile assegnare il valore `null` ad un oggetto della classe wrapper. Ma bisogna tener presente che `null` non è un valore valido per un tipo di dato primitivo. Quindi il seguente codice:

```
Boolean b = null;
boolean bb = b;
```

compila tranquillamente, ma lancia una `NullPointerException` al runtime. Questo è forse il principale problema che si può incontrare utilizzando l'autoboxing e l'autounboxing.

12.6.1.3 Costrutti del linguaggio ed operatori relazionali

Tutti i costrutti del linguaggio (`if`, `for`, `while`, `do`, `switch` e l'operatore ternario) e tutti gli operatori di confronto (`<`, `>`, `==`, `!=`, etc.) si basano sui tipi booleani. Con Java potremo sostituire un tipo `boolean` con il rispettivo tipo wrapper `Boolean` senza problemi, anche in tutti i costrutti e operatori. Il costrutto `switch` (cfr. paragrafi 4.3.4 e 10.3.6) inoltre, si basa su una variabile di test che può essere di tipo enumerazione, `String`, `byte`, `short`, `int` o `char`. Alla luce di quanto appena visto quindi, tale variabile potrà essere anche di tipo `Byte`, `Short`, `Integer` o `Character`. L'operatore `==` è utilizzabile per confrontare sia tipi di dati primitivi, sia reference. In ogni caso va a confrontare i valori delle variabili coinvolte che, nel caso di tipi reference, sono indirizzi in memoria (cfr. capitolo 4). Questo implica che due oggetti confrontati con l'operatore == saranno uguali se e solo se risiedono allo stesso indirizzo, ovvero se sono lo stesso oggetto. L'introduzione della doppia feature di autoboxing e autounboxing ci obbliga ad alcune riflessioni. Visto che i tipi primitivi e i tipi wrapper sono equivalenti, che risultati darà l'operatore = = ? Il risultato è quello che ci si aspetta sempre, le regole non sono cambiate. I valori dei reference rimangono i valori degli indirizzi in memoria, ma c'è un'eccezione. Consideriamo il seguente codice:

```
public class Comparison {
    public static void main(String args[]) {
        Integer a = 1000;
        Integer b = 1000;
        System.out.println(a==b);
        Integer c = 100;
        Integer d = 100;
        System.out.println(c==d);
        int e = 1000;
        Integer f = 1000;
        System.out.println(e==f);
        int g = 100;
        Integer h = 100;
        System.out.println(g==h);
    }
}
```

Il suo output sarà:

```
false
true
true
true
```

Tranne che nella comparazione tra `c` e `d`, tutto funziona in maniera normale. Infatti `c` e `d` sono due `Integer` con valori compresi nel range del tipo `byte` (`-128` a `127`) e quindi, come abbiamo asserito nel paragrafo precedente, sono trattati in maniera speciale dalla JVM. Questi due `Integer` vengono inscatolati nelle stesse istanze wrapper immutabili, per essere memorizzate e riusate. Quindi i due `Integer` risultano condividere lo stesso indirizzo perché in realtà sono lo stesso oggetto. L'ultima considerazione è da considerarsi indubbiamente un problema, ma se scriveremo codice che utilizza il metodo `equals()` con i tipi complessi e l'operatore `==` con i tipi primitivi, questo problema non si presenterà mai. In fondo, questa dovrebbe essere la regola da seguire sempre.

12.6.1.4 Overload:

L'overload (cfr. capitolo 8) è una caratteristica di Java molto semplice e potente. Grazie ad essa è possibile far coesistere in una stessa classe più metodi con lo stesso nome ma con differente lista di parametri. Prima

dell'avvento dell'autoboxing era molto semplice capire la chiamata ad un metodo sottoposto a overload. La lista dei parametri non generava nessun dubbio. Ma consideriamo il seguente codice:

```
public void metodo(Integer i) { . . . }
public void metodo(float f) { . . . }
```

Quale metodo verrà chiamato dalla seguente istruzione?

```
metodo(123);
```

La risposta è semplice ma può sembrare sorprendente: lo stesso che veniva chiamato con le versioni precedenti (ovvero `metodo(float f)`). In questo modo è stata evitata una scelta che avrebbe avuto conseguenze inaccettabili per gli sviluppatori, ovvero sarebbe stata negata la compatibilità con il codice preesistente.

12.6.2 Altri metodi delle classi wrapper

I metodi che consentono di fare il parsing da una stringa a un certo tipo di dato sono fondamentali. Per esempio `Integer` ha il metodo statico `parseInt()`, che tenta di tradurre in intero una stringa passata in input. Per esempio:

```
int i = Integer.parseInt("1");
```

recupera il valore `1` dalla stringa `"1"`. Siccome abbiamo a che fare con le stringhe bisogna stare attenti ed eventualmente gestire la `NumberFormatException`. Esistono anche i metodi `parseLong()` nella classe `Long`, `parseDouble()` nella classe `Double` e così via. In queste classi esistono ancora tantissimi altri metodi da sfruttare, tra cui il metodo `compareTo()` che abbiamo già trovato nella classe `String`. Infatti le classi wrapper implementano l'interfaccia funzionale e generica `Comparable`, a cui dedicheremo il paragrafo 12.8.1.

Con l'avvento di Java 8 le classi `Integer`, `Byte`, `Short`, `Long`, `Character`, `Double` e `Float` sono state arricchite con i metodi statici `sum()`, `max()` e `min()`, che però vedremo nel capitolo 17 dedicato alle collections e agli stream. Stesso discorso nella classe `Boolean` alla quale sono stati aggiunti

i metodi `logicalAnd()`, `logicalOr()` e `logicalXor()`. Inoltre è stata anche introdotta la possibilità di usare i **tipi senza segno (unsigned)**, tramite il metodo `toUnsignedInt()` e `toUnsignedLong()` per le classi `Byte` e `Short` e `Integer`. Per esempio:

```
byte b = (byte)(128);
System.out.println(Byte.valueOf(b));
System.out.println(Byte.toUnsignedInt(b));
```

stamperà:

```
-128
128
```

Il perché il metodo `valueOf()` stampi `-128` è già stato spiegato nel paragrafo 3.2.1, quando abbiamo parlato di tipi primitivi e loro rappresentazione. Invece con il metodo `toUnsignedInt()` stiamo usando l'intervallo degli 8 bit del tipo `byte`, usando solo i numeri positivi (senza segno), e quindi l'intervallo di rappresentazione cambia da `[-128, 127]` a `[0, 255]`. La classe `Integer` dichiara anche il metodo `toUnsignedString()`, che restituisce sotto forma di stringa il parametro intero in input. È come un metodo `parseInt()` senza segno. Di questo metodo ne esiste anche una versione con un secondo parametro intero, che permette di specificare la base di rappresentazione del numero intero. Per esempio, col seguente frammento di codice useremo le basi 10 (quella di default), 2, 8 e 16:

```
int i = (int)3_000_000_000L;
System.out.println(Integer.toUnsignedString(i,10));
System.out.println(Integer.toUnsignedString(i,2));
System.out.println(Integer.toUnsignedString(i,8));
System.out.println(Integer.toUnsignedString(i,16));
```

Questo stamperà le rappresentazioni del valore `3000000000` (che è fuori dell'intervallo di rappresentazione di un intero con segno) in formato decimale, binario, ottale ed esadecimale:

```
3000000000
10110010110100000101111000000000
26264057000
b2d05e00
```

Le classi `Integer` e `Long` inoltre dichiarano metodi per fare operazioni unsigned come `divideUnsigned()`, `remainderUnsigned()` (resto della divisione) e `compareUnsigned()`.

> **Per fare operazioni senza segno bisogna avere bene in mente come si rappresentano in formato binario i numeri negativi (tecnica del complemento a due spiegata nel paragrafo 3.2.3). In linguaggi come il C/C++ le operazioni senza segno sono un argomento complesso che è spesso causa di grossi problemi. Java ha fatto a meno dei tipi senza segno per quasi vent'anni proprio per evitare gli stessi problemi. L'esigenza è dovuta alla difficoltà di usare formati di file e protocolli di rete che richiedono l'aritmetica senza segno, quindi non è un argomento che useremo spesso in futuro.**

12.7 La classe `Math`

Questa classe appartiene al package `java.lang` (da non confondere con il package `java.math`). Ha la caratteristica di contenere solo membri statici: due costanti ("pi greco" e la "e" base dei logaritmi naturali) ed ottanta metodi statici. I metodi rappresentano le seguenti funzioni matematiche: valore assoluto (`abs()`), tangente (`tan()`), logaritmo (`log()`), potenza (`pow()`), massimo (`max()`), minimo (`min()`), seno (`sin()`), coseno (`cos()`), esponenziale (`exp()`), radice quadrata (`sqrt()`), arrotondamenti per eccesso (`ceil()`), per difetto (`floor()`) e classico (`round()`), generazione di numeri casuali (`random()`). Questa classe ha solo un costruttore privato, quindi non si può istanziare. Non è nemmeno possibile estenderla perché è dichiarata `final`. Un esempio di utilizzo della classe `Math` è il seguente:

```
System.out.println(Math.floor(5.6));
```

In questo caso sarà stampato il valore `5.0`. Infatti la "funzione" `floor()` restituisce un numero `double` che arrotonda per difetto il valore del parametro in input sino al più vicino numero intero.

Con Java 8 sono stati introdotti nuovi metodi del tipo "exact". Questi

metodi lanciano un'eccezione nel caso siano sforati i limiti numerici del tipo (overflow). Questo significa che se per esempio `100*1000000000` in Java restituisce il valore errato `1215752192` (cfr. capitolo 3), il metodo `Math.multiplyExact(100,1000000000)`, lancerà un'eccezione:

```
Exception in thread "main" java.lang.ArithmeticException:
integer
  overflow
      at java.lang.Math.multiplyExact(Math.java:867)
      at TestMathExact.main(TestMathExact.java:4)
```

Per arginare lo stesso problema esistono anche i metodi `addExact()`, `subtractExact()`, `negateExact()`, `multiplyExact()`, `incrementExact()` e `decrementExact()`.

12.8 Ordinamento

Con il termine **ordinamento**, intendiamo la disposizione di oggetti in base ad una sequenza crescente o decrescente, rispetto a determinati criteri, all'interno di una collezione. In particolare quando per esempio aggiungiamo elementi ad un array list, questi saranno disposti nell'ordine in cui sono stati aggiunti. Se invece volessimo ordinarli secondo altri criteri, dobbiamo definire un metodo che si occupi di questo. Come vedremo di seguito, questo metodo potrà trovarsi o all'interno degli oggetti da ordinare, o all'interno di un oggetto creato appositamente per ge stire l'ordinamento. Come esempio di ordinamento possiamo immaginare di voler ordinare un array list di oggetti `Persona`, secondo il criterio alfabetico dei loro cognomi come primo criterio, secondo il criterio alfabetico dei loro nomi come secondo criterio e così via.

12.8.1 L'interfaccia `Comparable`

L'ordinamento in Java richiede la definizione di un metodo che gestisca i criteri di ordinamenti di oggetti. L'interfaccia funzionale `Comparable` definisce un unico metodo astratto: `compareTo()`. È implementata da centinaia di classi della libreria standard. Abbiamo già incontrato infatti il metodo `compareTo()` più volte in questo capitolo, quando abbiamo esplorato i metodi della classe `String`, e delle classi wrapper. Infatti sia `String` che le

classi wrapper implementano l'interfaccia `Comparable`, che è anche generica. Ecco la sua definizione epurata dai commenti:

```
package java.lang;
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Per esempio la classe `String` è stata definita nel seguente modo:

```
public final class String implements Comparable<String>
//...
```

Il fatto di aver utilizzato nella definizione della classe `String` come tipo parametro per `Comparable` il tipo `String`, implica che il metodo `compareTo()` accetterà solo stringhe. Con la stessa logica sono state definite tutte le classi che implementano l'interfaccia `Comparable` (comprese le classi wrapper) per garantire che l'ordinamento sarà fatto solo tra oggetti istanziati dalla stessa classe. Quindi il metodo `compareTo()`, viene chiamato su un oggetto e gli viene passato un oggetto della stessa classe, e quindi il confronto all'interno del metodo viene fatto tra l'oggetto corrente e l'oggetto passato in input. Il **contratto** del metodo `compareTo()`, dice che il metodo deve ritornare un valore negativo se l'oggetto corrente è minore dell'oggetto in input. Deve ritornare `0` se i due oggetti sono identici. E deve tornare un valore positivo nel caso l'oggetto corrente sia maggiore dell'oggetto in input. Quindi il seguente frammento di codice:

```
Integer int1 = 4;
Integer int2 = 5;
Integer int3 = 5;
System.out.println(int1.compareTo(int2));
System.out.println(int2.compareTo(int1));
System.out.println(int2.compareTo(int3));
```

produrrà il seguente output:

```
-1
1
0
```

Ma come e quando si usa l'ordinamento? In generale quando una classe

implementa `Comparable`, allora vuol dire che è ordinabile all'interno di una collezione o un array. Per esempio se vogliamo ordinare un array di stringhe, avremo a disposizione il metodo statico `sort()` della classe di utilità `Arrays` (che si trova nel package `java.util`):

```
String [] nomi = {"Marcello","Serena","Antonio"};
System.out.println("Array non ordinato:");
stampaArray(nomi);
Arrays.sort(nomi);
System.out.println("\nArray ordinato:");
stampaArray(nomi);
```

dove il metodo `stampaArray()` stampa con un ciclo foreach gli elementi dell'array. Il codice precedente produrrà il seguente output:

```
Array non ordinato:
Marcello
Serena
Antonio

Array ordinato:
Antonio
Marcello
Serena
```

> **Esiste anche la classe `java.util.Collections` che rappresenta una classe di utilità per le collections, come `Arrays` lo è per gli array. Queste classi contengono anche tanti altri metodi di utilità per lavorare su collezioni ed array. Torneremo sull'argomento nel capitolo 16.**

12.8.2 L'interfaccia `Comparator`

L'interfaccia `Comparator` è molto simile a `Comparable`, ma non appartiene al package `java.lang`, bensì al package `java.util`. Ciónonostante ci sembra opportuno completare il discorso sull'ordinamento in questo paragrafo. L'interfaccia `Comparator` è anch'essa funzionale e generica. Segue la sua dichiarazione semplificata:

```
package java.util;
public interface Comparator<T> {
    int compare(T o1, T o2);
//...
```

La differenza tra il metodo `compare()` di `Comparator` e il metodo `compareTo()` di `Comparable`, è che `compare()` viene chiamato su un'istanza di `Comparator` e prende in input le due istanze della classe (che deve corrispondere al tipo generico) che bisogna comparare. Per esempio potremmo creare una classe che implementa `Comparator` che ordina le stringhe dopo averle rovesciate. In questo modo abbiamo definito un tipo di ordinamento diverso da quello naturale:

```
import java.util.Comparator;

public class StringComparator implements Comparator<String>
{
    public int compare (String s1, String s2) {
        StringBuilder sbl = new StringBuilder(sl);
        StringBuilder sb2 = new StringBuilder(s2);
        s1 = sb1.reverse().toString();
        s2 = sb2.reverse().toString();
        return s1.compareTo(s2);
    }
}
```

Si noti che abbiamo usato un oggetto il metodo `reverse()` per capovolgere le stringhe, dopo averle messe all'interno di istanze di `StringBuilder`, una classe d'utilità di `java.lang`, che effettua alcune operazioni interessanti sulle stringhe. Poi abbiamo sfruttato il metodo `compareTo()` per confrontare le stringhe rovesciate. A questo punto consideriamo per esempio il seguente frammento di codice:

```
List<String> nomiList = new ArrayList<>
(Arrays.asList(nomi));
System.out.println("List non ordinata:");
stampaList(nomiList);
Collections.sort(nomiList, new StringComparator());
System.out.println("List ordinata a stringhe al
contrario:");
stampaList(nomiList);
```

dove abbiamo usato il metodo generico `asList()` di `Arrays` per riempire la lista con gli elementi dell'array del precedente frammento di codice. Inoltre abbiamo usato un overload del metodo `sort()` che prende in input come parametro un oggetto `Comparator` che ci ha consentito di ordinare secondo il nuovo ordine stabilito dalla classe `StringComparator` creata da noi. L'output del precedente frammento di codice sarà:

```
List non ordinata:
Antonio
Marcello
Serena
List ordinata a stringhe al contrario:
Serena
Antonio
Marcello
```

Riepilogo

In questo capitolo abbiamo introdotto le classi principali del package `java.lang`, con qualche digressione su classi di altri package come **java.util** di cui abbiamo accennato alle classi `Objects`, `Arrays`, `Collections` e `Comparator`. La classe **Object** è particolarmente importante per il fatto che è la superclasse di tutte le classi, e quindi capita spesso di fare override dei suoi metodi. Il metodo **equals()** restituisce `true` solo se i due reference confrontati puntano allo stesso oggetto. Quindi solitamente questo metodo va ridefinito nelle sottoclassi. Insieme ad esso va ridefinito anche il metodo **hashcode()**, per supportare l'identificazione di un oggetto all'interno di collection basate su algoritmi di hash come `HashMap`. L'implementazione di default di quest'ultimo restituisce il numero esadecimale che rappresenta l'indirizzo in memoria dell'oggetto su cui è invocato. Abbiamo visto come invece esistono diverse strategie per generare efficienti hashcode. I tool di sviluppo supportano la generazione automatica di `equals()` e `hashcode()`. Altro metodo che va riscritto nelle sottoclassi è **toString()**, che ha il compito di restituire una stringa descrittiva dell'oggetto su cui viene chiamato. Nella sua implementazione di default nella classe `Object` restituisce una stringa di formato `nomeClasse@IndirizzoInEsadecimale`. Solitamente viene riscritto stampando il valore delle variabili d'istanza dell'oggetto. Anche in questo

caso i tool di sviluppo favoriscono la creazione automatica del metodo.

Il metodo **clone()** è inteso per clonare un oggetto. Nella sua implementazione di default realizza solo una *shallow copy*, ovvero vengono clonate le variabili d'istanza, ma se queste sono di tipo complesso, saranno copiati solo i reference e non gli oggetti. Per implementarlo in una sottoclasse è necessario sia implementare l'interfaccia `Cloneable`, sia definire nel metodo `clone()` una clausola `throws` all'eccezione `CloneNotSupportedException`, e bisogna anche modificare il modificatore del metodo da `protected` (nella classe `Object` è definito così) a `public`.

La classe **System** astrae il sistema su cui viene seguito il programma, non è istanziabile né estendibile. Definisce le variabili statiche **in, out** ed **err** che rappresentano rispettivamente lo standard input, lo standard output e lo standard error del sistema. Queste dipendono dal sistema su cui girano e possono essere reindirizzate verso altri input e output. La classe `System` ci permette anche di impostare e recuperare i valori delle **system properties**, ovvero coppie chiave-valore che rappresentano delle proprietà del sistema che possono essere recuperate in ogni parte del programma mediante il codice. Le system properties possono essere impostate sia dall'interno del programma che dall'esterno (da riga di comando o da file). `System` possiede anche altri metodi come **exit()** per far terminare istantaneamente il programma, **lineSeparator()** che restituisce il separatore di linee dipendente dal sistema operativo (e che corrisponde alla system property `line.separator`), **getEnv()** che restituisce le variabili d'ambiente del sistema, **currentTimeMillis()** che restituisce il numero di millisecondi passati a partire dal primo gennaio1970, e **arrayCopy()** che è capace di copiare un array in un altro.

Java 9 ha un nuovo algoritmo di default per deallocare memoria non più utilizzata basato sul garbage collector **G1.**

La classe **Runtime** definisce metodi per ottenere informazioni sul sistema e sulla memoria al runtime, ed anche il metodo **exec()** per eseguire programmi esterni. La **reflection** (o **introspezione delle classi)**, mediante la definizione della classe generica `Class` e le altre classi definite nel package `java.lang.reflect` come `Method` e `Field`, ci permette di leggere i dettagli di una classe compilata, ed anche di invocare i suoi costruttori e i suoi metodi. Ci sono tre modi per ottenere un'istanza della classe `Class`: chiamando il

metodo `getClass()` su un oggetto, utilizzando il metodo statico `forName()` della stessa classe `Class`, oppure utilizzando un **class literal.** La possibilità di creare oggetti a partire da stringhe (nome della classe) è particolarmente importante, e su di essa si basano molte tecnologie e framework moderni.

La classe **String** è dichiarata `final` e quindi non estendibile. Può essere istanziata senza utilizzare costruttore e operatore `new`, semplicemente con un literal. Quando avviene ciò, la virtual machine utilizza un **pool di stringhe** per minimizzarne l'utilizzo e riutilizzare le stringhe già istanziate. Definisce tantissimi metodi, nessuno dei quali è capace di modificare una stringa perché gli oggetti stringa sono **immutabili.** Nella versione 9 per la memorizzazione delle stringhe in memoria è stato adottato uno stratagemma noto come **Compact Strings**, che consente di risparmiare memoria e spesso di migliorare le prestazioni dei programmi. Le **classi wrapper** rappresentano un importantissimo strumento quando si utilizzano le collections e non solo. Le istanze delle classi wrapper sono anch'esse immutabili e in alcuni casi vengono trattate in memoria come le stringhe istanziate con la sintassi semplificata. Tra i metodi implementati ricordiamo soprattutto i metodi che convertono una stringa in un tipo wrapper come `Integer.parseInt()`. Con l'**autoboxing-autounboxing** però, è possibile utilizzare tipi wrapper e tipi primitivi quasi allo stesso modo. Nonostante questa funzionalità sia particolarmente semplice da utilizzare, nasconde comunque alcune insidie come per esempio quando si assegna un valore `null` ad un tipo wrapper, che non può essere gestito al runtime da un tipo primitivo.

La classe **Math** è non istanziabile e non estendibile. I suoi membri sono tutti statici e rappresentano utilità per compiere operazioni matematiche. Infine abbiamo anche affrontato il discorso dell **'ordinamento**, che può essere realizzato mediante l'implementazione dell'interfaccia funzionale **Comparable**, da implementare direttamente nella classe degli oggetti da ordinare, o alternativamente dall'implementazione dell'interfaccia funzionale **java.util.Comparator** che può essere implementata in una classe qualsiasi.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Conoscere l'utilizzo e i metodi più importanti della classe `Object` (unità 12.1)	☐	
Conoscere l'utilizzo e i metodi più importanti della classe `System` (unità 12.2)	☐	
Conoscere l'utilizzo e i metodi più importanti della classe `Runtime` (unità 12.3)	☐	
Conoscere l'utilizzo e i metodi più importanti della classe `Class` e saper sfruttare la tecnica della reflection (unità 12.4)	☐	
Conoscere l'utilizzo e i metodi più importanti della classe `String` (unità 12.5)	☐	
Conoscere l'utilizzo delle classi wrapper (unità 12.6)	☐	
Comprendere le semplificazioni che ci offre la (doppia) funzionalità di autoboxing e autounboxing (unità 12.6)	☐	
Conoscere l'utilizzo e i metodi più importanti della classe `Math` (unità 12.7)	☐	
Conoscere le interfacce `Comparable` e `Comparator` per ordinare elementi di array e collezioni (unità 12.8)	☐	

Note:

13

Tipi Annotazioni

Obiettivi:
Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Comprendere cosa sono i metadati e la loro utilità (unità 13.1, 13.2).
- ✔ Comprendere l'utilità delle annotazioni (unità 13.1, 13.2, 13.3).
- ✔ Saper definire nuove annotazioni (unità 13.1, 13.2).
- ✔ Saper annotare elementi Java ed altre annotazioni (unità 13.2, 13.3).
- ✔ Saper utilizzare le annotazioni definite dalla libreria: le annotazioni standard e le meta-annotazioni (unità 13.3, 13.4).
- ✔ Comprendere e saper utilizzare le repeating annotation e le type annotation (unità 13.2, 13.3).

Questo capitolo spiega un argomento sicuramente non semplice. La difficoltà di formulare correttamente le definizioni essenziali, la complessità di una sintassi non standard, il livello elevato di astrazione dal linguaggio, l'uso di una nuova parola chiave speciale, l'introduzione di un nuovo tipo di paradigma di programmazione e la relativa applicabilità delle annotazioni, richiederanno al lettore un livello di concentrazione elevato. Ma d'altronde le annotazioni rappresentano una delle caratteristiche più potenti di Java. In questo capitolo non parleremo di programmazione standard fatta da `if, for`

e paradigmi object oriented, ma di **meta-programmazione.** In ogni caso, riteniamo utile avvertire il lettore meno esperto che non sarà facile apprezzare alcuni aspetti di questo argomento. È improbabile per esempio che si definiscano da subito le proprie annotazioni e i propri processori di annotazioni. Esistono però alcune parti senz'altro più abbordabili in questo capitolo (vedi annotazioni standard) che potrebbero risultare utili nell'immediato.

In effetti abbiamo già incontrato un paio di annotazioni standard durante lo studio di questo libro. La prima è l'annotazione `Override`, studiata nel capitolo 8. Ricordiamo che con essa è possibile annotare un metodo che fa override di un metodo ereditato. In questo modo rendiamo consapevole il compilatore che abbiamo intenzione di fare un override. Esso sarà quindi capace di segnalarci un errore nel caso non trovasse nella superclasse un metodo con la stessa firma del metodo annotato con `Override`. Senza l'annotazione, rischieremo di creare involontariamente un metodo nuovo senza fare override, per poi accorgerci del problema solo al runtime. Per esempio se nella nostra classe definiamo il metodo `toString()` scrivendolo senza la "S" maiuscola:

```
public UnaClasseQualsiasi {
    //omettiamo il codice della classe
    public String tostring() { //dovremmo scrivere
"toString"
                               //e non "tostring"
       //omettiamo il codice del metodo
    }
}
```

avremo creato un metodo nuovo, e non un override del metodo della classe `Object`. In fase di esecuzione del programma ci accorgeremo che qualcosa non va, e se saremo attenti e intuitivi riusciremo ad individuare il problema (la "s" minuscola). È molto meglio annotare il metodo ed ottenere un errore in compilazione piuttosto che al runtime.

Un'altra annotazione che abbiamo incontrato è l'annotazione `SuppressWarning` a cui abbiamo accennato all'interno del paragrafo 9.5.1 dedicato ai warning. Annotando un metodo con `SuppressWarning` possiamo evitare che il compilatore ci segnali particolari tipologie di warning. Tutto molto semplice! In effetti molti programmatori Java reputano

quest’argomento banale e secondario. Basta limitarsi ad utilizzare le annotazioni standard e quelle definite nei framework come Java EE, Hibernate, Struts, etc. In fondo il programmatore le utilizza e un tool le processa, niente di più normale! Ma le annotazioni sono molto più di questo, perché possiamo crearle noi, e possiamo anche creare il tool che deve interpretarle.

La prima parte di questo capitolo sarà quindi dedicata alla definizione delle annotazioni. È abbastanza complessa, ma importante. Infatti impareremo a creare le nostre annotazioni e ne studieremo la sintassi. La seconda parte del capitolo è dedicata allo studio delle annotazioni standard, che potranno risultare utili da subito, anche al programmatore meno esperto. Ne approfitteremo per analizzare anche aspetti più complessi che potrebbero interessare meno al neofita, ma illuminare lo sviluppatore esperto.

13.1 Definizione di annotazione (metadato)

Con il termine **metadati**, solitamente si intendono le “informazioni sulle informazioni”. La frase precedente non è poi così chiara, ma dovremmo filosofeggiare un po’ per poter dare una definizione corretta. Quindi cercheremo di capire il concetto con esempi prima di entrare nel cuore dell’argomento. Nel capitolo 2 abbiamo definito il concetto di classe come “un insieme di oggetti che condividono le stesse caratteristiche e le stesse funzionalità”. Ma proviamo a guardare l’argomento da un’altra angolazione. Proviamo a pensare cos’è una classe per il compilatore o la virtual machine. Sarà più facile calandoci in un esempio. Se volessimo spiegare a qualcuno in lingua italiana cos’è una persona, cosa che potrebbe anche essere molto complessa, utilizzeremmo una serie di frasi del tipo “ha un nome, un cognome, un’età, etc.”. Definiremmo il concetto di persona tramite l’elenco delle sue caratteristiche e delle sue funzionalità. Tutto questo va bene, ma è comprensibile solo se si hanno già in mente le definizioni di nome, cognome, età e così via. Ovvero, se il nostro interlocutore non sa cosa è un nome, potrebbe pensare che un nome valido sia “831”. Ugualmente, se non ha mai sentito parlare di età, potrebbe pensare che un’età possa avere questo valore: “-99”. Ecco allora il bisogno dei metadati: le informazioni sulle informazioni. Se definiamo un’età come “un numero intero di anni non minore di zero”, diventa tutto più chiaro.

Per qualcuno l'esempio precedente potrebbe essere contestabile. Nel mondo reale non bisogna (fortunatamente) definire per forza tutti i metadati per farsi capire, almeno in una situazione così banale. Ma se volessimo definire cosa è per la geometria differenziale la "superficie romana di Steiner" (uno degli argomenti della mia tesi di laurea in matematica) non credo che senza una quantità notevole di metadati potremmo capirci.

Molto spesso i metadati non sono altro che **vincoli** (in inglese **constraints).** Per esempio, nella definizione di età abbiamo specificato come vincolo che il numero degli anni non può essere minore di zero. Un **vincolo** è considerato una delle definizioni più importanti per molte metodologie moderne. Esiste infatti anche una sintassi in UML per specificare vincoli (cfr. appendice L). Addirittura, la sintassi UML stessa è definita tramite un linguaggio che si basa proprio sul concetto di vincolo. Tale linguaggio infatti si chiama **Object Constraint Language (OCL).**

I metadati sono quindi informazioni su informazioni. Nel nostro esempio la definizione di età era un metadato relativo alla definizione di persona. Ma relativamente alla definizione di età, potrebbe essere un metadato la definizione di anno. Quindi il concetto di metadato è sempre relativo a ciò che si sta definendo. Inoltre, un metadato è anche relativo all'interlocutore che ne ha bisogno. Se per esempio spieghiamo ad un adulto un concetto come quello di persona, non dovrebbe essere necessario specificare metadati. Se il nostro interlocutore è un bambino, invece, la situazione cambia.

Se poi il linguaggio in cui bisogna astrarre il concetto di persona è Java, allora non avremo di fronte un interlocutore umano, ma solo un freddo software. Che questo software sia il compilatore, la Java Virtual Machine, Javadoc o JAR non cambia molto. Questi software non possono capire se stiamo definendo correttamente la classe `Persona`, a meno che non specifichiamo metadati.

Per le specifiche del linguaggio Java, una classe è infatti definita tecnicamente come "metamodello Java".

Per esempio, definiamo la classe `Persona` nel seguente modo:

```
public class Persona {
     //...
     private int anni;
     public void setAnni(int anni) {
          if (anni < 0)
             throw new IllegalArgumentException(anni
               + " anni, non è una età valida!";
             this.anni = anni;
          }
     }
     public int getAnni() {
          return anni;
     }
}
```

Con il controllo nel metodo `setAnni()` abbiamo specificato un vincolo destinato alla JVM. Un altro vincolo specificato dal codice precedente è invece riservato al compilatore: il modificatore `private`. Questo rappresenta un metadato che specifica al compilatore che la variabile `anni` non ha visibilità all'esterno della classe. Esistono in Java anche altri meccanismi per specificare metadati. Per esempio, potremmo specificare con il tag `@deprecated` interno ad un commento javadoc, che un certo metodo è deprecato. Questo tipo di metadato è invece destinato al tool javadoc (cfr. paragrafo 3.5.4). Con l'introduzione dei **tipi annotazioni**, o più semplicemente **annotazioni** (in inglese **type annotations)**, Java ha un modo standard per definire nuovi metadati di un programma. Per esempio è possibile specificare *vincoli* definiti dallo sviluppatore che possono essere *interpretati* da un software creato ad hoc che interpreta le annotazioni. Le annotazioni, essendo metainformazioni dirette verso un software, hanno un orizzonte colmo di possibilità.

In Java 8 è stato rimosso, insieme a tutte le relative librerie contenute nel package `com.sun.mirror`, lo strumento Annotation Processing Tool, meglio conosciuto come APT, presente invece nelle precedenti versioni.

13.1.1 Primo esempio

Tecnicamente le annotazioni sono un altro tipo di struttura dati del linguaggio, che si va ad aggiungere alle classi, alle interfacce e alle enumerazioni. Si dichiarano in maniera simile alle interfacce, ma si definiscono con una sintassi che sconfina dallo standard Java. Di seguito riportiamo un primo esempio di definizione di annotazione:

```
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

Come si può notare, la definizione di annotazione coinvolge una parola chiave che non abbiamo ancora incontrato: `@interface`. Questa serve proprio per definire un'annotazione come la parola chiave `class` serve a dichiarare una classe, o come `enum` serve a dichiarare un'enumerazione. In questo caso l'annotazione si chiama `DaCompletare` e definisce due strani metodi astratti (ma non sono marcati dal modificatore `abstract` perché è implicito come per le interfacce): `descrizione()` e `assegnataA()`. Questi due metodi, detti **elementi dell'annotazione** (in inglese **annotation elements)**, hanno una sintassi abbreviata equivalente alla dichiarazione di una variabile con relativo metodo omonimo che restituisce il suo valore, una sorta di metodo getter. Si noti anche come il metodo `assegnataA()` utilizzi anche la parola chiave `default`. Questa serve a specificare un valore di default per il metodo nel caso in cui, utilizzando l'annotazione, lo sviluppatore non fornisca un valore per la variabile invisibile `assegnataA`., implicitamente dichiarata. Ma approfondiremo tra poco i dettagli della sintassi.

> `DaCompletare` **viene detto quindi "tipo annotazione" ("annotation type").**

Dopo aver visto come si dichiarano le annotazioni, cerchiamo di capire come

si utilizzano. È possibile utilizzare un'annotazione come si fa con un modificatore, per esempio per un metodo, utilizzando la seguente sintassi:

```
public class Test {
    @DaCompletare(
        descrizione = "Bisogna fare qualcosa...",
        assegnataA = "Claudio"
    )
    public void faQualcosa() {

    }
}
```

> **Si noti che è anche possibile anteporre il modificatore (nell'esempio `public`) all'annotazione, non cambia nulla a parte la leggibilità. In quel caso `public` si riferirà sempre al metodo e non all'annotazione.**

Analizziamo brevemente la sintassi utilizzata. L'istanza dell'annotazione viene dichiarata come se fosse un modificatore del metodo `faQualcosa()`. La sintassi però è molto particolare. Si utilizza il simbolo di chiocciola `@` (si dovrebbe leggere "AT") che si antepone al nome dell'annotazione. Poi si aprono parentesi tonde per specificare i parametri, in modo simile a quanto si fa con un metodo. Ma specificare i parametri di un'annotazione significa specificare coppie del tipo "chiave = valore", dove le chiavi corrispondono alle variabili (mai dichiarate) relative al metodo omonimo. Quindi se nell'annotazione abbiamo definito i metodi `descrizione()` e `assegnataA()`, abbiamo anche in qualche modo definito le variabili invisibili (elementi) `descrizione` e `assegnataA`. Il loro tipo corrisponde esattamente al tipo di ritorno del metodo omonimo (in entrambi i casi avevamo definito il tipo `String`). Infatti tale metodo funzionerà da metodo getter (o accessor), ovvero restituirà il valore della relativa variabile.

Il metodo `faQualcosa()` è a questo punto stato annotato, ma manca ancora qualcosa. Bisogna creare un software che interpreti l'annotazione implementando un comportamento. Infatti, se noi possiamo anche intuire a cosa serve l'annotazione `DaCompletare`, nessun software avrà mai la nostra

stessa perspicacia. Per esempio, potremmo creare un'applicazione la quale riceva in input una classe, legga le eventuali annotazioni e pubblichi su una bacheca in Intranet i compiti da assegnare ai vari programmatori.

```
import java.lang.reflect.*;
import java.util.*;

public class AnnotationsPublisher {
    public static void main(String[] args) throws Exception
{
        Map<String, String> map = new HashMap<>();
        Method[] methods =
Class.forName("Test").getMethods();
        for (Method m :  methods) {
            DaCompletare dc = null;
            if ((dc = m.getAnnotation (DaCompletare.class))
!= null) {
                String descrizione = dc.descrizione();
                String assegnataA = dc.assegnataA();
                map.put(descrizione, assegnataA);
             }
        }
        publicaInIntranet(map);
    }

    public static void publicaInIntranet(Map <String,
String>map) {
        Set <String>keys = map.keySet();
        for (String key :  keys) {
            System.out.println("Descrizione = " + key
              + "; Assegnata a: " + map.get(key));
        }
    }
}
```

La classe `AnnotationsPublisher` dichiara una mappa parametrizzata con chiave e valore di tipo `String`. Con un ciclo foreach estrae, tramite reflection, i metodi della classe `Test`. Sfruttando i metodi della classe `Method`, ed in particolare il metodo `getAnnotation()`, vengono scelti solo i metodi annotati con `DaCompletare`. Per essi sono memorizzate nella mappa le informazioni. Finito il ciclo, viene invocato il metodo `pubblicaInIntranet()` che si occuperà di estrarre le informazioni dalla

mappa per pubblicarle in Intranet.

Questo era solo un esempio, ma le potenzialità delle annotazioni sono teoricamente illimitate. Per semplicità abbiamo omesso la prima parte della definizione dell'annotazione `DaCompletare`. Senza questa parte non sarà possibile far funzionare correttamente l'esempio. Segue la definizione completa dell'annotazione:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

Come è possibile notare l'annotazione è stata a sua volta annotata, e sarà spiegato tutto a partire dal paragrafo 13.2. Ma se è possibile annotare un metodo ed un'annotazione, cos'altro possiamo annotare? La lista è lunga: classi, attributi, parametri, variabili locali... praticamente tutto! È possibile anche decidere se un'annotazione deve essere destinata alla lettura del solo compilatore o può essere utilizzata in fase di runtime. Di default l'annotazione viene riportata nel file .class ma non è considerata dalla JVM. Nei prossimi paragrafi chiariremo tutti gli aspetti delle annotazioni.

13.1.2 Tipologie di annotazioni e sintassi

Come per le enumerazioni, anche le annotazioni saranno alla fine trasformate in classi dal compilatore. Questo è necessario per garantire la compatibilità del codice con le versioni precedenti di Java. Le classi compilate implementeranno l'interfaccia `java.lang.annotation.Annotation`. Si noti che, implementando "manualmente" tale interfaccia, non definiremo un'annotazione; si può dire che tra le annotazioni e l'interfaccia `Annotation` c'è lo stesso rapporto vigente tra le enumerazioni e la classe `java.lang.Enum`.

> **Un'annotazione non può utilizzare la parola chiave** `implements` **né la parola chiave** `extends`**. Quindi in nessun modo un'annotazione può estendere un'altra annotazione,**

una classe, o implementare un'interfaccia.

Possiamo distinguere tre tipologie di annotazioni: le **annotazioni ordinarie**, le **annotazioni a valore singolo** e le **annotazioni marcatrici.**

13.1.2.1 Annotazione ordinaria (o completa)

L'unica annotazione che abbiamo già visto è un esempio di **annotazione ordinaria.** Si tratta del tipo di annotazione più complesso. Infatti viene detto anche **annotazione completa** (in inglese **full annotation).** Rianalizziamo l'annotazione dell'esempio:

```
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

L'annotazione `DaCompletare` definisce due metodi che sono implicitamente pubblici e astratti. Infatti non è possibile aggiungere altri modificatori come `protected`, `private`, `static` o `final`, mentre è possibile (ma superfluo come per le interfacce) utilizzare i modificatori `abstract` e `public`. Come già asserito in precedenza però, questi metodi hanno un'implementazione implicita. Infatti, dichiarare un metodo equivale a dichiarare una coppia costituita da una variabile ed un metodo omonimi. Quest'ultimo restituisce il valore della variabile come farebbe un metodo getter. Questi metodi astratti sono quindi detti **elementi dell'annotazione.**

Un'altra particolarità nella sintassi è l'utilizzo della parola chiave `default` per assegnare un valore predefinito alla variabile nascosta `assegnataA`. Possiamo quindi immaginare che un'annotazione sia una specie d'interfaccia che viene implementata da una classe creata al volo dal compilatore, simile alla seguente:

```
public class DaCompletareImpl implements DaCompletare {
    private String descrizione;
    private String assegnataA = "da assegnare";
    public String descrizione() {
        return descrizione;
    }
```

```
    public String assegnataA() {
        return assegnataA;
    }
}
```

Per un elemento di un'annotazione non è possibile né specificare parametri di input né indicare `void` come valore di ritorno. Il compilatore segnalerà errori esplicitamente.

Come in un'interfaccia, è possibile dichiarare all'interno di un'annotazione, oltre a metodi (implicitamente astratti) anche costanti statiche, enumerazioni e in generale tipi innestati (cfr. capitolo 10). Tutto ciò che si dichiara in un'annotazione, che non sia un elemento di un'annotazione, è implicitamente `public`, `static` e `final`. Per esempio possiamo arricchire l'annotazione in questo modo:

```
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
    enum Priorita {ALTA, MEDIA, BASSA};
    Priorita priorita() default Priorita.ALTA;
}
```

La convenzione per gli identificatori delle annotazioni è identica a quella delle classi, delle interfacce e delle enumerazioni.

All'interno dell'annotazione potremmo anche aggiungere costanti statiche o classi innestate, come:

```
int a = 0;
class ClasseInnestata {
    //implementazione omessa
}
```

Inutile utilizzare modificatori per questi elementi che sono automaticamente dichiarati `public`, `static` e `final`.

Dopo aver studiato la sintassi della dichiarazione di un'annotazione, analizziamo ora la sintassi del suo utilizzo. Un'annotazione, come visto nell'esempio precedente, viene utilizzata come se fosse un modificatore, e per convenzione precede gli altri modificatori, ma non è obbligatorio. La sintassi per modificare un elemento di codice Java (classe, metodo, variabile locale, parametro, etc.) è sempre del tipo:

```
@NomeAnnotazione ([lista di coppie] nome=valore)
```

Nell'esempio precedente avevamo annotato un metodo nel seguente modo:

```
@DaCompletare(
    descrizione = "Bisogna fare qualcosa...",
    assegnataA = "Claudio"
)
public void faQualcosa() {

}
```

È obbligatorio impostare tutte le variabili i cui metodi non dichiarano un valore di `default`. Quindi è legale scrivere:

```
@DaCompletare(
    descrizione = "Bisogna fare qualcosa..."
)
public void faQualcosa() {

}
```

Mentre il seguente codice:

```
@DaCompletare(
    assegnataA = "Claudio"
)
public void faQualcosa() {

}
```

provocherà un errore in compilazione:

```
Test.java:4: annotation DaCompletare is missing descrizione
  assegnataA = "Claudio"
               ^
```

```
1 error
```

Se invece consideriamo l'annotazione `DaCompletare` nella versione arricchita dall'enumerazione, allora potremmo utilizzarla nel seguente modo:

```
@DaCompletare(
    descrizione = "Bisogna fare qualcosa...",
    priorita = DaCompletare.Priorita.BASSA
)
public void faQualcosa() {

}
```

> **Avendo fornito un `default` anche al metodo `priorita()`, è possibile ometterne l'impostazione.**

13.1.2.2 Annotazione a valore unico

Il secondo tipo di annotazione è detto **annotazione a valore unico** (in inglese **single value annotation).** Si tratta di un'annotazione contenente un unico elemento di annotazione che viene chiamato `value()`. È possibile dichiarare un qualsiasi tipo di ritorno per il metodo `value()` tranne il tipo `java.lang.Enum`. Per esempio, la seguente annotazione è di tipo a valore unico:

```
public @interface Serie {
    Alfabeto value();
    enum Alfabeto {A, B, C};
}
```

Infatti l'unico elemento di annotazione si chiama `value()`, mentre l'enumerazione innestata `Alfabeto` non si può considerare un elemento dell'annotazione. Anche la seguente è un'annotazione a valore unico:

```
public @interface SingleValue {
    int value();
}
```

La seguente annotazione invece:

```
public @interface SerieOrdinaria {
    Alfabeto alfabeto();
    enum Alfabeto {A,B,C};
}
```

è un'annotazione ordinaria, perché non definisce come unico elemento `value()`. La differenza con altri tipi di annotazioni sta nella sintassi dell'utilizzo. Per utilizzare un'annotazione a valore unico è infatti possibile scrivere all'interno delle parentesi solamente il valore da assegnare. Per esempio, in luogo di:

```
@Serie(value = Serie.Alfabeto.A)
public void faQualcosa() {

}
```

potremmo scrivere più semplicemente:

```
@Serie(Serie.Alfabeto.A)
public void faQualcosa() {

}
```

senza specificare quale elemento dell'annotazione stiamo impostando (visto che sappiamo che è unico). Il risultato è lo stesso.

13.1.2.3 Annotazione marcatrice

La terza tipologia di annotazioni è chiamata **annotazione marcatrice** (in inglese **marker annotation)**, a volte anche detta **annotazione segnalibro.** Si tratta della tipologia più semplice: un'annotazione che non ha elementi. Per esempio, il seguente è un esempio di annotazione segnalibro:

```
public @interface Marker {}
```

Questo tipo di annotazioni vengono utilizzate con la sintassi che ci si aspetterebbe:

```
@Marker()
public void faQualcosa() {

}
```

e possono risultare molto più utili di quanto non ci si aspetti. È anche possibile evitare di specificare le parentesi tonde, come mostra il seguente codice:

```
@Marker public void faQualcosa() {

}
```

> **Un esempio di annotazione segnalibro è l'annotazione** `Override`.

Le annotazioni si possono anche innestare nelle classi, come nel seguente esempio:

```
public class Test {
    public @interface Serie {
        Alfabeto value();
        enum Alfabeto {
            A,B,C
        };
    }
    @Serie(Serie.Alfabeto.A)
    public void faQualcosa() {

    }
}
```

13.2 Annotare annotazioni (meta-annotazioni)

Nel package `java.lang.annotation` sono definite sei **meta-annotazioni.** Si chiamano `Retention`, `Target`, `Documented`, `Inherited`, `Native` e `Repeatable`. Queste ultime due sono state introdotte solo con Java 8. Tutte queste annotazioni servono solo ad annotare altre annotazioni.

Abbiamo già visto un esempio di meta-annotazione nel paragrafo precedente. Infatti, dopo aver presentato il primo esempio, abbiamo evidenziato come l'annotazione `DaCompletare` per funzionare correttamente doveva essere annotata a sua volta dall'annotazione `Retention` nel seguente

modo:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

In questo caso passando a `Retention` il valore unico `RetentionPolicy.RUNTIME`, abbiamo specificato che l'annotazione `DaCompletare` era destinata alla lettura da parte del runtime Java (la lettura della JVM). Senza l'utilizzo della meta-annotazione, l'annotazione `DaCompletare` sarebbe stata inclusa nel bytecode di default dopo la compilazione, ma ignorata dal runtime Java. L'annotazione `Retention` è dettagliata nel paragrafo 13.2.2.

13.2.1 Target

La prima meta-annotazione che affronteremo è `java.lang.annotation.Target`. Il suo scopo è specificare gli elementi del linguaggio a cui è applicabile l'annotazione che si sta definendo. Infatti, in italiano `Target` significa "obiettivo". Questa meta-annotazione è di tipo *a valore singolo*, e prende come parametro un array di `java.lang.annotation.ElementType`. `ElementType` è un'enumerazione definita come segue:

```
package java.lang.annotation;

public enum ElementType {
    TYPE, // Classi, interfacce, o enumerazioni
    FIELD, // variabili d'istanza (anche se enum)
    METHOD, // Metodi
    PARAMETER, // Parametri di metodi
    CONSTRUCTOR, // Costruttori
    LOCAL_VARIABLE, // Variabili locali o clausola catch
    ANNOTATION_TYPE, // Tipi Annotazioni
    PACKAGE, // Package
    TYPE_PARAMETER, // Tipi parametro (visti con i Generics)
    TYPE_USE, // Uso di un tipo
```

```
    MODULE //Moduli di Java 9
}
```

Essa specifica, con i suoi elementi, i vari elementi del linguaggio Java a cui è possibile applicare un'annotazione. Per esempio potremmo utilizzare questa meta-annotazione per limitare l'applicabilità dell'annotazione `DaCompletare`:

```
import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;

@Target({TYPE, METHOD, CONSTRUCTOR, PACKAGE,
ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

Adesso l'annotazione `DaCompletare` è applicabile solo a classi, interfacce, enumerazioni, annotazioni, metodi, costruttori e package. Si noti come gli elementi dell'array di `ElementType` siano specificati mediante la sintassi breve dell'array (cfr. paragrafo 3.6.3). Inoltre tali elementi sono stati utilizzati senza essere referenziati grazie all'import statico. Notiamo inoltre che, se si vuole passare come parametro a `Target` un unico elemento, è possibile utilizzare anche la sintassi:

```
@Target(TYPE);
```

Il compilatore in questo caso è capace di capire le nostre intenzioni, e non ci obbliga ad utilizzare un array quando usiamo un solo elemento.

> **Se definiamo un'annotazione senza utilizzare la meta-annotazione `Target`, la nostra annotazione sarà applicabile di default a tutti gli elementi di tipo dichiarazione, tranne `TYPE_PARAMETER` e `TYPE_USE`.**

Può risultare interessante come viene definita l'annotazione `Target`:

```
package java.lang.annotation;
```

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    ElementType[] value();
}
```

`Target` viene annotata anche da sé stessa, affinché sia applicabile solo alle annotazioni.

13.2.1.1 Type Annotations

Una novità importante che ha portato Java 8, è la possibilità di usare l'elemento dell'enumerazione `ElementType.TYPE_USE`. Prima dell'avvento di Java 8 era possibile annotare solo dichiarazioni, ora anche un qualsiasi **uso di tipi** (si parla di **Type Annotations).** Per esempio, ora è possibile annotare tipi quando si istanziano, nella clausola `implements`, nei cast, nella clausola `throws` e così via. Per esempio, supponendo che esista la seguente annotazione:

```
import java.lang.annotation.*;
@Target(ElementType.TYPE_USE)
@interface TestTP {
}
```

Allora è possibile scrivere:

```
public class TypeParameterExample<@TestTP T> {
    public void metodo(@TestTP T t) {
        System.out.println(t);
    }
}
```

Per ora non esistono nella libreria ufficiale annotazioni con questo tipo di `Target`, ma l'Università di Washington ha rilasciato il suo Checker Framework, che ne fa ampio uso. Per esempio in quest'ambizioso progetto è definita l'annotazione `@NotNull`, che può essere usata ogni volta che vogliamo che una certa variabile non debba diventare `null`. Questo framework è capace di interpretare il codice e capire come evitare la più frequente delle eccezioni al runtime: la `NullPointerException`.

Indipendentemente dal fatto se userete o meno il Checker Framework, è importante notare quanto può essere potente utilizzare le type annotation. Nel caso dell'annotazione `@NotNull`, si sta cercando di eliminare uno dei più grandi problemi della programmazione Java, e questo è davvero notevole. Per chi fosse interessato al Checker Framework, è possibile consultare il seguente link: http://types.cs.washington.edu/checker-framework.

Seguono alcuni esempi d'uso di type annotations all'interno di una classe molto complicata da leggere, ma che copre quasi tutti i casi di utilizzo:

```
import java.io.Serializable;
import java.util.*;
public class TPExamples<T extends @TestTP List<T>> extends
@TestTP Object
  implements @TestTP Serializable {
    private static final long serialVersionUID = 1L;
    public static void main(String args[])throws @TestTP
Exception {
        byte i =(@TestTP byte)1000;
        Object o = new @TestTP String();
        if (o instanceof @TestTP String) {
           String s = (@TestTP String)o;
        }
        List<@TestTP String> list = null;
        List<? extends @TestTP Serializable> list2 = null;
        @TestTP String array @TestTP[] @TestTP[];
    }
}
```

Si può notare l'uso della type annotation `@TestTP` già nella dichiarazione della classe. La classe è generica ed ha un tipo parametro bounded il cui limite è annotato. Inoltre si noti l'utilizzo dell'annotazione nella clausola `implements` e nella clausola `extends`. Anche il metodo `main()` è dichiarato con la clausola annotata `throws`. Poi l'annotazione viene usata in un cast di primitivi (variabile `i`) e in un cast di tipi complessi (variabile `s`), nonché all'interno di un controllo con `instanceof`. Poi viene annotato anche un tipo parametro (della variabile `list`), ed un altro tipo parametro bounded wildcard. Infine un array bidimensionale viene annotato interamente (variabile `array[] []`), ed anche i suoi sotto-array (`array[i]` e `array[j]`, dove `i` e `j` rappresentano le righe e le colonne a cui solitamente pensiamo per convenzione). In definitiva si possono usare in quasi tutte le situazioni in cui

si usa un tipo primitivo o complesso. In particolare non si possono usare negli `import`, né con i class literals. Per esempio è illegale scrivere:

```
import java.util.@TestTP List;
```

e anche:

```
@TestTP List.class
```

non compilerà.

13.2.2 Retention

La meta-annotazione `Retention` (che in italiano possiamo tradurre come "conservazione") è anch'essa molto importante. Serve per specificare come deve essere conservata dall'ambiente Java l'annotazione a cui viene applicata. Come `Target`, anche `Retention` è di tipo a valore singolo, ma prende come parametro un valore dell'enumerazione `java.lang.annotation.RetentionPolicy`. Segue la definizione di `RetentionPolicy`, con commenti esplicativi sull'uso dei suoi valori:

```
package java.lang.annotation;
public enum RetentionPolicy {
    SOURCE, // l'annotazione è eliminata dal compilatore
    CLASS, /* l'annotazione viene conservata anche nel file
              ".class", ma ignorata dalla JVM */
    RUNTIME /* l'annotazione viene conservata anche nel file
               ".class", e letta dalla JVM */
}
```

Come già affermato precedentemente la meta-annotazione `Retention` applicata alla annotazione `DaCompletare` farà in modo che le annotazioni di tipo `DaCompletare` siano conservate nei file compilati, per essere infine letti anche dalla virtual machine. Riportiamo nuovamente il codice di seguito:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

Nel paragrafo 13.1.1 denominato “Primo esempio” infatti, lo scopo di questa annotazione era essere letta tramite reflection al runtime, e quindi era obbligatorio specificare la meta-annotazione.

13.2.3 Documented

Questa semplice meta-annotazione ha il compito di includere nella documentazione generata da Javadoc anche le annotazioni a cui è applicata. Per esempio, la meta-annotazione `Target` è a sua volta annotata da `Documented`, come mostra la sua dichiarazione:

```
package java.lang.annotation;
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    ElementType[] value( );
}
```

Questo significa che, se generiamo la documentazione del nostro codice tramite il comando `javadoc` sui file dove è utilizzato `Target`, sarà riportata anche l’annotazione. Per esempio, se generiamo la documentazione della seguente annotazione:

```
import java.lang.annotation.*;
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface MaxLength {
    int value();
}
```

che specifica la lunghezza massima di un certo campo (e che si può verificare a runtime), avremo come risultato quanto mostrato in figura 13.1

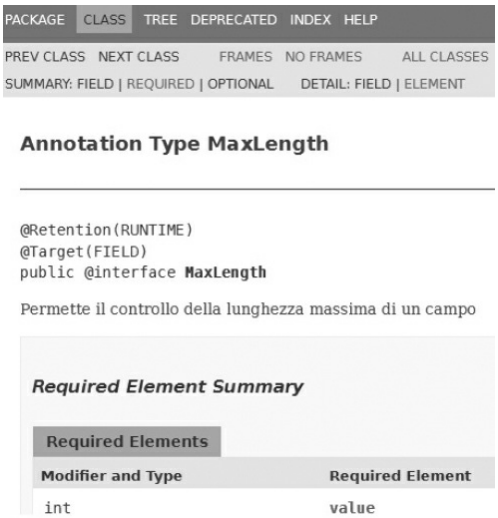


Figura 13.1 - Un particolare della documentazione generata.

> **In figura 13.1 è possibile notare come, essendo anche `Retention` annotata con `Documented`, essa viene documentata.**

13.2.4 Inherited

La meta-annotazione `Inherited` permette alle annotazioni applicate a classi (e solo a classi) di essere ereditate. Ciò significa che se abbiamo la seguente annotazione:

```
import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;
@Target({TYPE, METHOD, CONSTRUCTOR, PACKAGE,
ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

annotata a sua volta da `Inherited`, e l'applichiamo alla seguente classe:

```
@DaCompletare (
    descrizione = "Da descrivere..."
```

```
)
public class SuperClasse {
    //...
}
```

che a sua volta viene estesa dalla seguente sottoclasse:

```
public class SottoClasse extends SuperClasse {
    //...
}
```

anche quest'ultima sarà annotata allo stesso modo della superclasse. Basterà eseguire il codice seguente per verificarlo:

```
import java.lang.reflect.*;
import java.util.*;
import java.lang.annotation.*;
public class AnnotationsReflection2 {
    public static void main(String[] args) throws Exception
{
        Annotation[] dcs =
SottoClasse.class.getAnnotations();
        for (Annotation dc :  dcs) {
            System.out.println(dc);
       }
    }
}
```

> **Le annotazioni annotate con `Inherited` sono ereditate solo se applicate a classi. Tali annotazioni, se applicate a metodi, interfacce, o qualsiasi altro elemento Java che non sia una classe, non saranno ereditate.**

13.2.5 Repeatable

A volte potrebbe essere necessario annotare più volte con la stessa annotazione un elemento. Per esempio, potremmo dichiarare un'annotazione chiamata `TestTrigger`, che potrebbe essere letta da un eventuale software esterno, al fine di ottenere informazioni su come e quando fare il test di una certa classe. Per esempio:

```
@TestTrigger(quando="Ogni giorno, ore 18", come =
  TestTrigger.StrumentoDiTest.JUNIT)
@TestTrigger(quando="Ogni venerdì, ore 9", come =
  TestTrigger.StrumentoDiTest.GUI)
public class TestRepeatable {
    public void metodo(String... args) {
        //...
    }
}
```

Abbiamo annotato con la stessa annotazione `TestTrigger` (che spiegheremo tra poco) la classe `TestRepeatable`. Prima di Java 8 questo tipo di sintassi avrebbe portato ad un errore in compilazione. Il programmatore avrebbe dovuto creare un'annotazione "contenitore" a mano, che permettesse di fatto di usare più volte l'annotazione. Ora invece, con l'introduzione della meta-annotazione `Repeatable`, è possibile dichiarare annotazioni che si possono applicare anche più di una volta. Ma Java ha da sempre sposato il principio della compatibilità con le versioni precedenti. Se questo da un lato ha portato a creare qualche disagio con le nuove funzionalità che venivano introdotte (vedi **erasure** per i generics introdotti in Java 5), ha di fatto permesso ai programmi di aggiornare la versione di Java senza grossi traumi. Anche in questo caso, per permettere le cosiddette **repeating annotations** (in italiano **annotazioni che si ripetono)** si è dovuto scendere a un compromesso: l'annotazione contenitore viene creata dal compilatore in base alle nostre direttive. Per creare una repeating annotation, bisogna per prima cosa annotare la nostra annotazione con `Repeateable` come nel seguente esempio:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@Documented
@Repeatable(TestTriggers.class)
public @interface TestTrigger {
    String quando();
    StrumentoDiTest come() default StrumentoDiTest.JUNIT;
    public enum StrumentoDiTest {
        JUNIT, GUI, JMETER, SOAPUI;
    }
}
```

L’annotazione `TestTrigger` contiene una enumerazione innestata `StrumentoDiTest`, che definisce i diversi modi di *come* deve essere eseguito il test. Inoltre definisce due elementi: `come` proprio di tipo `StrumentoDiTest`, e `quando` che invece è di tipo stringa. Questa annotazione `TestTrigger` è annotata da tutte le meta-annotazioni che abbiamo incontrato sinora, tra cui anche `Repeatable`. Quest’ultima è un’annotazione a valore unico che prende in input il tipo dell’annotazione contenitore (`TestTriggers`, si noti il plurale) che il compilatore deve creare. Per fare ciò, dobbiamo almeno dichiarare l’annotazione che rappresenta il tipo contenitore, altrimenti il codice precedente non compila. Quindi creiamo l’annotazione che farà da contenitore `TestTriggers`:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@Documented
public @interface TestTriggers {
    TestTrigger[] value();
}
```

Molto semplicemente deve essere un’annotazione a valore unico che ritorna un array di `TestTrigger`. È possibile leggere tramite reflection le diverse annotazioni usate, usando il metodo `getAnnotationByType(Class<T>)` dell’interfaccia `AnnotatedElement`, passandogli l’annotazione contenitore. Per esempio:

```
Class<TestRepeatable> testRepeatableClass =
TestRepeatable.class;
TestTriggers [] annotations =

testRepeatableClass.getAnnotationsByType(TestTriggers.class);

for (TestTriggers testTriggers: annotations) {
    TestTrigger[] values = testTriggers.value();
    for(TestTrigger testTrigger: values) {
       System.out.println(testTrigger.quando());
       System.out.println(testTrigger.come());
    }
}
```

In questo esempio abbiamo solo stampato i valori di `come()` e `quando()`, ma possiamo immaginare che sarebbe stato utile creare con una logica complessa dei metodi che interpretano questi valori, per poi schedulare i test per la classe annotata `TestRepeatable`. Per esempio:

```
public static void interpretaQuando(String quando) {
    // codice complesso che interpreta la stringa
    // e programma l'esecuzione del test
}
public static void
interpretaCome(TestTrigger.StrumentoDiTest come) {
    // codice complesso che registra lo strumento
    // di test per questo caso
}
```

Oppure è possibile usare il metodo `getAnnotations(Class<T>)` per farsi ritornare le occorrenze di utilizzo delle annotazioni tutte assieme.

13.3 Annotazioni standard

Le annotazioni che ci accingiamo a studiare sono dette **annotazioni standard.**

In particolare le annotazioni `Override`, `FunctionalInterface`, `Deprecated` e `SuppressWarning` sono state già nominate precedentemente tra le pagine di questo libro. Le prime tre sono molto usate, mentre le annotazioni `Native` (non ancora utilizzata) e `SuppressWarning` sono utilizzate più raramente.

13.3.1 Override

Come abbiamo già visto precedentemente, l'annotazione `java.lang.Override` può essere utilizzata per indicare al compilatore che un metodo in una classe è un override di un altro metodo della sua superclasse. Di seguito è riportata la sua dichiarazione:

```
package java.lang;
import java.lang.annotation.*;
@Target(value=ElementType.METHOD)
@Retention(value=RetentionPolicy.SOURCE)
public @interface Override {}
```

Come è possibile notare si tratta di un'annotazione di tipo marcatrice, quindi non si devono specificare valori quando la si utilizza. Inoltre è applicabile solo a metodi (si noti il valore della meta-annotazione `Target`). Infine, tramite il valore di `Retention` impostato a `RetentionPolicy.SOURCE`, viene specificato che `Override` è una annotazione interpretabile solo dal compilatore e che non sarà inserita nel bytecode relativo. L'utilità di questa annotazione è piuttosto intuitiva. Per esempio, un tipico errore del programmatore Java abbastanza difficile da scoprire, è quello di fare un override sbagliando il nome del metodo nella sottoclasse. Usando questa annotazione dichiariamo al compilatore che stiamo consapevolmente facendo un override. Il compilatore quindi potrà eventualmente segnalarci un problema nel caso il nome del metodo non sia scritto correttamente. Inoltre aggiunge maggiore espressività al nostro codice: con un solo colpo d'occhio sapremo che il metodo che stiamo riscrivendo non è un metodo qualunque, ma uno riscritto da una superclasse. Il baco del nome sbagliato diventa ancora più difficile da risolvere quando sbagliamo un override estendendo una classe di un framework, dove il metodo riscritto deve essere chiamato dal framework stesso, e non da codice che scriviamo noi. Per esempio per scrivere un'applet (un'applicazione Java che gira all'interno di un browser ma deprecata con Java 9, cfr. appendice Q) bisogna estendere la classe `java.applet.Applet`, e riscriverne i metodi ereditati `init()` e `start()`. Siccome in un'applet non esiste il metodo `main()`, questi metodi rappresentano il ciclo di vita dell'applicazione stessa. Quindi quando viene caricata l'applet, viene prima chiamato il metodo `init()`, poi `start()` e così via. Basterebbe scrivere:

```
public void Init() {
    //...
}
```

(si noti la lettera maiuscola) per perdere molto tempo a capire perché l'applet non viene inizializzata. Invece se annotiamo questo metodo con `@Override`, il problema viene evidenziato direttamente a livello di compilazione.

Come già asserito in precedenza, il compilatore eseguito con l'opzione -Xlint **segnalerà dei warning nel caso in cui**

troverà nel nostro codice override non annotati adeguatamente.

13.3.2 Deprecated

L'annotazione standard `java.lang.Deprecated` serve per indicare al compilatore e al runtime di Java che un metodo o un qualsiasi altro elemento di codice Java è deprecato. Ricordiamo che gli elementi in Java vengono deprecati allo scopo di segnalare agli utenti che lo stesso elemento non si dovrebbe più usare perché sostituito da qualche altro elemento nella sua funzionalità, e che in futuro potrebbe essere rimosso.

In generale quando si depreca un elemento, gli si dovrebbe aggiungere anche un commento per specificare cosa sostituisce l'elemento deprecato.

Quindi, l'annotazione `Deprecated` ha per il compilatore la stessa funzione che il tag `@deprecated` ha per l'utility Javadoc. Sono due costrutti complementari e vanno utilizzate contemporaneamente. Infatti se utilizzassimo solo il tag javadoc `@deprecated`, il compilatore ci restituirebbe un warning simile al seguente:

```
warning: [dep-ann] deprecated item is not annotated with
@Deprecated
```

La versione di Java 9 ha ridefinito il tag `Deprecated` trasformandolo da tipo marker ad annotazione ordinaria. Infatti ora dichiara due elementi opzionali come si può vedere dalla attuale dichiarazione:

```
import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;
```

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD,
PACKAGE,
               MODULE, PARAMETER, TYPE})
public @interface Deprecated {
    String since() default "";
    boolean forRemoval() default false;
}
```

Il primo dei due nuovi elementi si chiama `since` (che tradurremo come "esiste da"), e consente di specificare la versione del programma a partire dalla quale l'elemento annotato va considerato deprecato. Come valore di default ha stringa vuota. Il secondo elemento invece si chiama `forRemoval`, e serve per indicare se l'elemento annotato sarà rimosso in una futura versione del programma. Questo elemento ha come valore di default `false`. È possibile utilizzare questa annotazione con tutti i tipi definiti dall'annotazione `Target` tranne `TYPE_USE`, `ANNOTATION_TYPE` e `TYPE_PARAMETER`. Il seguente codice rappresenta un esempio di utilizzo di `Deprecated`:

```
@DaCompletare (
    descrizione = "Da descrivere..."
)
public class SuperClasse {
   /**
   * Questo metodo è stato deprecato
   * @deprecated utilizza un altro metodo per favore
   */
   @Deprecated (since="1.1", forRemoval="true")public void
metodo() {
       //...
   }
}
```

Le opzioni del compilatore `-deprecation` e `-Xlint:deprecation` sono equivalenti. Se viene per esempio utilizzato il metodo di `SuperClasse`, otterremo in fase di compilazione un warning simile al seguente:

```
TestAnnotation.java:4: warning: [deprecation] metodo() in
  SuperClasse has been deprecated
    sc.metodo();
      ^
```

```
1 warning
```

sia che venga esplicitata l'opzione `-deprecation`, sia che venga utilizzata l'opzione `-Xlint:deprecated`.

> **Java 9 ha anche introdotto un nuovo tool del JDK chiamato** jdeprscan, **che permette di eseguire scansioni su** .class, **file JAR e cartelle contenenti file** .class, **per evidenziare l'uso di elementi deprecati.**

13.3.3 FunctionalInterface

L'annotazione `FunctionalInterface` è stata introdotta con Java 8. È dichiarata nel seguente modo:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

Si tratta di una interfaccia marcatrice ed è molto semplice intuire che il suo scopo sia quello di annotare una **functional interface**, ovvero un'interfaccia che ha un unico metodo astratto (ma potrebbe avere tanti metodi statici, privati o di default). Essendo annotata con `@Documented`, usare questa annotazione significa anche esplicitare sulla documentazione javadoc che l'interfaccia annotata sia un'interfaccia funzionale. Non è obbligatorio usare questa annotazione affinché un'interfaccia sia considerata funzionale. Un'interfaccia si dice **interfaccia funzionale** se ha solamente un metodo che è astratto. Usando però quest'annotazione per l'interfaccia, il compilatore potrà segnalarci eventuali errori. Potrebbe capitare per esempio che una certa interfaccia sia dichiarata con un unico metodo astratto originariamente, ma che poi nel tempo qualcuno la vada a modificare aggiungendoci un altro metodo astratto. In casi come questo l'annotazione farà sì che il compilatore segnali un errore. Per esempio la compilazione della seguente interfaccia:

```
@FunctionalInterface
public interface FunctionalInterfaceExample {
    void metodoAstratto();
```

```
    void secondoMetodoAstratto();
    default void metodoDiDefault(){
        System.out.println("metodo di default");
    }
 }
```

darà luogo al seguente errore:

```
FunctionalInterfaceExample.java:1: error:
  Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
  FunctionalInterfaceExample is not a functional interface
    multiple non-overriding abstract methods found in
interface
      FunctionalInterfaceExample
1 error
```

Come vedremo le interfacce funzionali sono molto importanti in Java, perché sono alla base della teoria delle espressioni lambda.

13.3.4 SuppressWarnings

Come abbiamo evidenziato più volte sino ad ora, capiterà che compilando il nostro codice saranno evidenziati dei warning dal compilatore. Per esempio, il seguente frammento di codice:

```
 public static void main(String args[]) {
    List strings = new ArrayList<String>();
    strings.add("Lambda");
    //...
    Iterator<String> i = strings.iterator();
    while (i.hasNext()) {
        String string = i.next();
        System.out.println(string);
    }
 }
```

avrà un esito di compilazione positivo, ma provocherà il seguente output:

```
Note: TestSuppressWarnings.java uses unchecked or unsafe
operations.
Note: Recompile with -Xlint:unchecked for details.
```

Tali warning (che nelle specifiche vengono definiti **lint)** avvertono lo sviluppatore che esistono operazioni non sicure o non controllate, e viene richiesto di ricompilare il file con l'opzione `-Xlint` per avere ulteriori dettagli. Seguendo il suggerimento del compilatore, ricompiliamo il file con l'opzione richiesta. L'output sarà:

```
TestSuppressWarnings.java:5: warning: [rawtypes] found raw
type: List
        List strings = new ArrayList<String>();
        ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
TestSuppressWarnings.java:6: warning: [unchecked] unchecked
call
  to add(E) as a member of the raw type List
        strings.add("Lambda");
                    ^
  where E is a type-variable:
    E extends Object declared in interface List
TestSuppressWarnings.java:10: warning: [unchecked] unchecked
conversion
        Iterator<String> i = strings.iterator();
                                              ^
  required: Iterator<String>
  found: Iterator
3 warnings
```

Sono stati segnalati tre warning. Il primo è di tipo rawtypes, visto che abbiamo utilizzato `List` senza specificare il tipo parametro `String`. Gli altri due sono di tipo unchecked, perché la collezione `strings` viene aggiornata con il metodo `add()` senza che essa sia parametrizzata. Inoltre l'iteratore `i` viene usato con il tipo parametro `String`, senza che il compilatore possa essere certo che la lista `strings` contenga stringhe, visto che non è stato dichiarato il tipo parametro.

Ignorare i warning, solitamente non è saggio. Ma esistono casi in cui si preferisce lasciare il codice così com'è, magari perché è funzionante e non si vuole rischiare di creare bug. In tali casi è possibile utilizzare l'annotazione `SuppressWarnings`. Questa infatti può svolgere il ruolo di modificatore per classi, metodi, costruttori, variabili d'istanza, moduli, parametri e variabili

locali, affinché non generino warning. Si tratta questa volta di un'annotazione a valore unico, il cui parametro è di tipo array di stringhe. Serve per specificare la tipologia di warning Xlint (per esempio unchecked) che deve essere soppressa dal compilatore. Segue la sua dichiarazione:

```
package java.lang;
import java.lang.annotation.*;
import java.lang.annotation.ElementType;
import static java.lang.annotation.ElementType.*;
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR,
         LOCAL_VARIABLE, MODULE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

Per esempio, se vogliamo che il compilatore non generi warning né di tipo unchecked né di tipo rawtypes per il metodo dell'esempio precedente, sarà sufficiente annotare tale metodo nel seguente modo:

```
@SuppressWarnings({"unchecked", "rawtypes"})
public static void main(String args[]) {
    List strings = new ArrayList<String>();
    strings.add("Lambda");
    //...
    Iterator<String> i = strings.iterator();
    while (i.hasNext()) {
        String string = i.next();
        System.out.println(string);
    }
}
```

In particolare, tale annotazione consente di sopprimere warning riguardanti l'elemento annotato. Se l'elemento annotato contiene anche altri elementi che possono provocare warning (dello stesso tipo specificato) anche questi saranno soppressi dal compilatore. Per esempio, se una classe è annotata per sopprimere i warning di tipo deprecated, saranno soppressi eventuali altri warning dello stesso tipo relativi ai metodi della classe. Come consiglio stilistico proveniente direttamente da Joshua Block (lo sviluppatore di tale annotazione), i programmatori dovrebbero sempre annotare l'elemento più innestato. È sconsigliato annotare una classe per annotare tutti i suoi metodi.

Sarebbe meglio annotare ogni metodo.

> **Se si specifica come parametro due volte la stessa stringa come valore (per esempio due volte** unchecked), **la seconda occorrenza sarà ignorata. Saranno anche ignorate tutte le occorrenze non valide per la sintassi** Xlint.

13.3.5 Native

Tra i vari modificatori che abbiamo incontrato sino ad ora, ce n'è uno di cui non abbiamo ancora parlato: il modificatore `native`. Questa annotazione appartiene al package `java.lang.annotation`. È stato deciso infatti di creare un'appendice apposita per la sua spiegazione, perché considerato un modificatore usato piuttosto raramente. Questo da la possibilità di far interagire codice Java con codice scritto in linguaggio nativo, ovvero in C/C++. Per maggiori dettagli consultare l'appendice on line M. L'annotazione `Native`, appartiene al package `java.lang.annotation`, ed è così dichiarata:

```
@Documented
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.SOURCE)
public @interface Native {
}
```

È quindi una annotazione marcatrice applicabile solo alla dichiarazione di attributi ed in particolare costanti. La sua funzione è quella di indicare che la costante annotata può essere referenziata da codice nativo. Un software scritto appositamente potrebbe generare file header (usati in C/C++) e in base a questa annotazione determinare se e come crearli.

Riepilogo

I **metadati** sono informazioni su informazioni. Un**'annotazione** è un metadato sul codice Java. È possibile annotare praticamente tutti i componenti della programmazione. In questo modo le annotazioni possono

essere lette ed interpretate da software che può leggere il nostro codice tramite reflection. Possiamo **dichiarare annotazioni** grazie ad una sintassi particolare che si basa sulla parola chiave `@interface`. In un'annotazione è possibile dichiarare gli **elementi dell'annotazione**, ovvero metodi pubblici implicitamente astratti che sottintendono la definizione implicita di una variabile omonima, per cui svolgono il ruolo di metodo di accesso. Agli elementi dell'annotazione è possibile dare un **valore di default** grazie alla parola chiave `default`. In un'annotazione è possibile dichiarare anche elementi innestati (interfacce, classi o enumerazioni) e costanti statiche. In particolare tutto ciò che è dichiarato all'interno di un'annotazione che non è un elemento dell'annotazione, è automaticamente dichiarato `public`, `static` e `final`. Possiamo **usare annotazioni** anteponendole agli elementi di programmazione che vogliamo annotare. Nel caso le annotazioni usate dichiarino elementi che non hanno un valore di default, bisogna specificare un valore al momento dell'uso. Esistono tre tipi di annotazioni. L**'annotazione marcatrice** non dichiara elementi, e può essere utilizzata senza specificare le parentesi tonde che solitamente sono utilizzate per includere le impostazioni degli elementi stessi. L**'annotazione a singolo valore**, dichiara un unico elemento di annotazione chiamato `value()`. Si può utilizzare specificando tra le parentesi tonde direttamente il valore per l'elemento `value` (ma senza citarlo). L**'annotazione ordinaria** invece è un'annotazione che, o dichiara un unico elemento ma questo non si chiama `value()`, o dichiara più di un elemento. Le **meta-annotazioni** sono annotazioni che possono annotare altre annotazioni. Grazie alla meta-annotazione `Target` possiamo annotare un'annotazione per specificare a quali elementi di programmazione questa potrà essere applicata. L'annotazione `Retention` invece ci permette di annotare annotazioni al fine di specificare come deve essere conservata dall'ambiente Java l'annotazione a cui viene applicata. Può essere conservata a livello di codice, a livello di compilazione o a livello di runtime. La meta-annotazione `Documented` ha il compito di includere nella documentazione generata da Javadoc anche le annotazioni a cui è applicata. L'annotazione `Inherited` serve per annotare annotazioni, che, quando applicate a classi (e non a metodi, variabili, interfacce etc.) saranno ereditate dalle sottoclassi. L'annotazione `Repeatable` consente di annotare un'annotazione affinché essa possa annotare più di una volta un

certo elemento. Tali annotazioni sono dette **repeatable annotations** (ovvero **annotazioni ripetibili).** Per fare ciò però, c'è bisogno anche di dichiarare un'annotazione ausiliare a valore unico, che rappresenti la molteplicità dell'annotazione ripetibile. Le **annotazioni standard** sono quelle dichiarate nella libreria standard. L'obiettivo dell'annotazione **Override**, è quello di annotare i metodi che fanno override di un metodo ereditato. In questo modo, oltre ad avere maggiore leggibilità del codice, faremo in modo che il compilatore ci avverta nel caso l'override non sia implementato correttamente. L'annotazione **Deprecated** serve per annotare tipi, costruttori, metodi, parametri, moduli, package, variabili d'istanza e locali, per avvertire il compilatore (ed altri software) che il componente annotato è deprecato. L'annotazione **FunctionalInterface** permette di annotare le interfacce che devono essere interfacce funzionali. Un**'interfaccia funzionale** deve avere un unico metodo astratto. Se annotiamo un'interfaccia con `FunctionalInterface` che non è una vera interfaccia funzionale, il compilatore ci restituirà un errore. L'annotazione **SuppressWarning**, ci consente di annotare elementi come classi, metodi, costruttori, variabili d'istanza, moduli, parametri e variabili locali. Una volta annotati è possibile specificare una o più categorie di warning da far ignorare al compilatore. Infine l'annotazione **Native** può annotare solo costanti. Ha il compito di indicare che la costante annotata può essere referenziata da codice nativo. Un software scritto appositamente potrebbe generare file header (usati in C/C++) e in base a questa annotazione determinare se e come crearli.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere cosa sono i metadati e la loro utilità (unità 13.1, 13.2)	□	

Comprendere l’utilità delle annotazioni (unità 13.1, 13.2, 13.3)	☐	
Saper definire nuove annotazioni (unità 13.1, 13.2)	☐	
Saper annotare elementi Java ed altre annotazioni (unità 13.2, 13.3)	☐	
Saper utilizzare le annotazioni definite dalla libreria: le annotazioni standard e le meta-annotazioni (unità 13.3, 13.4)	☐	
Comprendere e saper utilizzare le repeating annotation e le type annotation (unità 13.2, 13.3)	☐	

Note:

14

Le librerie di utilità: il package `java.util` e Date-Time API

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Saper implementare programmi configurabili mediante file di properties e preferences (unità 14.1).
- ✔ Saper implementare programmi con l'internazionalizzazione (unità 14.1).
- ✔ Saper utilizzare la classe `StringTokenizer` per suddividere stringhe (unità 14.1).
- ✔ Saper utilizzare le basi delle espressioni regolari (unità 14.1).
- ✔ Comprendere a grandi linee le classi `Date` e `Calendar` (unità 14.1).
- ✔ Comprendere e saper utilizzare la nuova libreria Date-Time API (unità 14.2).

Questo capitolo è dedicato alla parte della libreria standard che definisce le classi di utilità. In particolare introdurremo alcune classi del package `java.util` (appartenente al modulo `java.base`) e dettaglieremo la libreria nota come **Date-Time API** ("API" è l'acronimo di "Application

Programming Interface" che tradurremo semplicemente con "libreria"). Quest'ultima, introdotta nella versione 8 di Java, si pone l'obiettivo di sostituire il vecchio modo di gestire le date e gli orari con le storiche classi `Date`, `Calendar` e `GregorianCalendar`. Questo capitolo non coprirà tutte le circa 200 classi presenti in questo package, la documentazione ufficiale delle API di Java deve sempre rimanere il riferimento principale. Cercheremo piuttosto di capire come utilizzare le classi e le interfacce principali. In particolare impareremo a far interagire i nostri programmi con file di configurazione, ad internazionalizzare i contenuti (ovvero fare in modo che i nostri programmi possano supportare più lingue), ad analizzare e formattare le stringhe, ad usare le espressioni regolari, e a gestire date e orari.

14.1 II package `java.util`

Il package `java.util` è stato creato per fornire classi di utilità. Un buon esempio di classe di utilità è la classe `Objects`, che offre metodi di utilità per oggetti generici. In questo package troviamo anche altre classi di utilità per array (la classe `Arrays`) e per le collection (classe `Collections`). Abbiamo già incontrato queste classi nel capitolo 12, insieme all'interfaccia funzionale `Comparator`, che ci ha fornito un modo alternativo rispetto all'uso dell'interfaccia `java.lang.Comparable`, per ordinare gli elementi in array e collezioni. Ma `java.util` contiene tante altre classi che sono argomento di questo capitolo.

In questo package vengono definiti anche altri tipi di cui parleremo nei prossimi capitoli. Per esempio viene definita la classe `Scanner` che consente di leggere i caratteri digitati sulla tastiera, che per coerenza spiegheremo nel capitolo 18 dedicato all'Input e Output. Ma anche `Timer` e `TimerTask` che permettono di creare dei timer appartengono a questo package, e verranno presentate nel prossimo capitolo sui Thread. Il package `java.util` soprattutto definisce il framework Collections, di cui abbiamo già usato più volte classi come `ArrayList` e `HashMap`. Tale framework sarà affrontato in dettaglio nel capitolo 17, insieme alla libreria "Stream API", ed altre classi di utilità.

In questo caso, con il termine "framework" intendiamo un insieme di classi e di interfacce riutilizzabili ed estendibili.

Quindi il package `java.util`, è un package a cui non è possibile dedicare un solo capitolo. Le sue classi ed interfacce forniscono soluzioni in contesti eterogenei, e in questo capitolo ci concentreremo solo su alcune di esse.

14.1.1 Le classi `Properties` e `Preferences`

La classe `Properties` rappresenta un insieme di proprietà di tipo chiave-valore che possono essere salvate in un file. Estende la classe `Hashtable` (un'implementazione dell'interfaccia `Map` del framework Collections, molto simile ad `HashMap` introdotta nel paragrafo 10.1.5) e ne eredita i metodi. Rispetto ad un `HashMap`, un oggetto `Properties` può leggere e salvare tutte le coppie chiave-valore in un file (detto **file di properties)** mediante i metodi `load()` e `store()`. Solitamente un file di properties si usa per memorizzare le informazioni per configurare la propria applicazione, ovvero si usa come **file di configurazione.** Come esempio prendiamo i file sorgenti di EJE (disponibili per il download all'indirizzo http://sourceforge.net/projects/eje). Il file di properties chiamato EJE_options.properties è utilizzato per leggere e salvare le preferenze dell'utente come la lingua, la versione di Java, la dimensione dei caratteri e così via. Per spiegare l'esempio ci concederemo un excursus su qualche classe del package `java.io`, che poi approfondiremo nel capitolo 18. Con il seguente codice (file EJE.java), mediante un blocco statico, viene chiamato il metodo `loadProperties()`:

```
static {
    try {
        properties = new Properties();
        try {
            loadProperties();
            //...
        } catch (FileNotFoundException e) {
            //...
        }
    }
}
```

La variabile `properties` era stata dichiarata come variabile d'istanza. Segue la dichiarazione (semplificata) del metodo `loadProperties()`:

```
public static void loadProperties() throws
FileNotFoundException {
    try (FileInputStream inputStream =
      new
FileInputStream("resources/EJE_options.properties");) {
        options.load(inputStream);
        //...
    } catch (IOException e) {
        //...
    }
}
```

L’oggetto `FileInputStream` che implementa la classe `InputStream` (e in questo esempio ne sfrutta anche un reference) permette di leggere il contenuto di un file. In questo caso al costruttore di `FileInputStream` abbiamo passato il file EJE_options.properties della cartella resources, che è un semplice file di testo al cui interno ci sono righe nel formato:

```
chiave=valore
```

Eccone un estratto:

```
eje.java.assertions=true
eje_area.braces_style=Java Style
eje.javac.target=9
file.recent8=D:\\java9\\Codice\\modulo_05\\esempi\\Punto.java

eje.javac.version=9
file.recent7=D:\\java9\\Codice\\modulo_07\\esempi\\Punto.java

eje_area.class_wizard=true
user.work_dir=;D:\\java9\\Test;D\:\\java8\\Codice;
file.recent6=D:\\java9\\Codice\\modulo_11\\esempi\\Punto.java

file.recent5=D:\\java9\\Codice\\modulo_11\\esempi\\PuntoHashCod

file.recent4=D:\\java9\\Codice\\modulo_12\\esempi\\AnnotationsR

file.recent3=D:\\java9\\Codice\\modulo_12\\esempi\\DaCompletare

file.recent2=D:\\java9\\Codice\\modulo_12\\esempi\\SottoClasse.

file.recent1=D:\\java9\\Codice\\modulo_12\\esempi\\SuperClasse.
```

```
eje_area.font.size=14
eje.java.docs=C\:\\Program Files\\Java\\jdk-9.0.1\\docs
eje_area.tab=4
eje.style=javax.swing.plaf.metal.MetalLookAndFeel
project.classpath=.;
eje.jdk.path=C\:\\Program Files\\Java\\jdk-9.0.1
project.output.path=
eje.lang=en
file.number.recent=8
project.docs.path=./docs
eje.java.version=9
eje_area.font.style=1
eje_area.font.type=SansSerif
```

Chiamando il metodo `load()` sull'oggetto `properties`, quest'ultimo viene riempito sfruttando il formato delle sue righe. Quindi senza sforzi la nostra mappa sarà già pronta per l'uso. Per esempio per leggere la lingua usata da EJE, la proprietà con chiave eje.lang può essere letta con la seguente sintassi:

```
String language = properties.getProperty("eje.lang");
```

Nell'editor EJE, dopo che è stato aperto un file, il suo nome viene salvato nella lista dei file recenti ed è impostato come primo file recente (dopo aver fatto slittare tutti gli altri di un posto). Per fare questo si imposta la seguente proprietà nell'oggetto `properties`:

```
properties.setProperty("file.recent1", fileName);
```

dove la variabile `fileName` è il nome del file che è stato aperto in EJE. Poi viene chiamato il seguente metodo:

```
public static void saveProperties() {
    OutputStream os = null;
    try {
        os = new
FileOutputStream("resources/EJE_options.properties");
        properties.store(os, "EJE OPTIONS - DO NOT EDIT");
    } catch (FileNotFoundException e) {
      //...
    }
}
```

Come l'oggetto `FileInputStream` ci ha permesso di leggere dal file, il

`FileOutputStream` ci consente di scrivere all'interno del file. Con il metodo `store()` a cui passiamo l'oggetto `os`, e una stringa che farà da testata (in inglese "header") del file, salviamo nuovamente tutte le coppie chiave=valore riga per riga all'interno del file.

> **Il file di properties a volte viene salvato dopo averlo modificato a mano. Nel caso un valore di una certa chiave sia molto lungo, e lo si desideri riportare su più righe, bisogna utilizzare il simbolo di slash "/" alla fine di ogni riga quando si vuole andare a capo. Per esempio:**
> ```
> eje.message=This message is very very very very /
> very long
> ```

Gestire file di properties è quindi molto semplice. È anche possibile sfruttare i metodi `loadFromXml()` e `saveToXML()` per gestire semplici file di properties in formato XML.

> **Un'introduzione a XML è contenuta nell'appendice on line H. Il supporto di Java a XML sarà invece approfondito nell'appendice on line N denominata "Java e XML".**

Una classe che offre funzionalità simili a `Properties`, è la classe `Preferences` del package `java.util.prefs`. La differenza essenziale è che con la classe `Preferences` le coppie chiave-valore non vengono inserite in un file specificato, bensì sono memorizzate in maniera dipendente dalla piattaforma. Insomma se usiamo questa classe potremo modificare la configurazione dell'applicazione solo attraverso l'applicazione stessa, e mai a mano. L'utilizzo è abbastanza semplice e il seguente esempio dovrebbe testimoniarlo (leggere i commenti):

```
import java.util.prefs.Preferences;

public class TestPreferences {
    private Preferences preferences;
    private final static String key1 = "key1";
```

```
    private final static String key2 = "key2";
    private final static String key3 = "key3";

    public TestPreferences() {
        // Istanza di un oggetto Preferences.
        preferences = Preferences.userRoot();
    }

    public void putPreferences() {
        // Settiamo i valori con diversi tipi
        preferences.putBoolean(key1, false);
        preferences.put(key2, "Pluto");
        preferences.putInt(key3, 100);
    }

    public void printPreferences() {
        // Stampiamo il valore di key1, se non lo troviamo
        // stampiamo il default true.
        System.out.println(preferences.getBoolean(key1,
true));
        // Stampiamo il valore di key2, se non lo troviamo
        // stampiamo il default pippo.
        System.out.println(preferences.get(key2, "pippo"));
        // Stampiamo il valore di key3, se non lo troviamo
        // stampiamo il default 0.
        System.out.println(preferences.getInt(key3, 0));
    }

    public void removePreferences() {
        // rimuoviamo il valore di key3
        preferences.remove(key3);
    }

    public static void main(String[] args) {
        TestPreferences test = new TestPreferences();
        test.putPreferences();
        test.printPreferences();
        test.removePreferences();
        test.printPreferences();
    }
}
```

In questa classe dichiariamo tre costanti statiche (`key1`, `key2`, `key3`) e un oggetto di tipo `preferences`. Nel costruttore di questa classe andiamo ad

istanziare l'oggetto sfruttando il metodo statico `userRoot()` della classe `Preferences`. In particolare è possibile istanziare oggetti di tipo `Preferences` a più livelli, come nei nodi di un file XML. Infatti avremmo potuto anche ottenere un oggetto `Preferences` su un altro nodo con un'istruzione come la seguente:

```
preferences =
Preferences.userRoot().node(this.getClass().getName());
```

In questo modo avremmo istanziato l'oggetto `preferences` su di un nodo figlio (a cui abbiamo passato il nome della classe `TestPreferences`), dove sarebbero state immagazzinate le coppie chiave-valore. La possibilità di poter definire diversi nodi, è una delle principali differenze che ne caratterizzano la flessibilità rispetto agli oggetti `Properties`. Per esempio potremmo creare nodi diversi per immagazzinare proprietà che appartengono ad aree funzionali diverse dell'applicazione.

Il metodo `putPreferences()`, imposta i valori delle coppie chiavi valori, sfruttando, oltre al classico metodo `put()`, anche metodi basati sul tipo come `putInt()` e `putBoolean()`. Il metodo `removePreferences()` invece rimuove la coppia la cui chiave era `key3`. Infine il metodo `printPreferences()` stampa i valori delle coppie dell'oggetto `preferences`. Si noti come i metodi getter definiti dalla classe `Preferences` consentano di specificare anche un valore di default da ritornare nel caso la coppia richiesta non sia trovata. L'output di questa classe sarà:

```
false
Pluto
100
false
Pluto
0
```

Infatti, dopo la rimozione del valore di `key3`, viene stampato il valore `0`.

Possiamo notare come, la creazione di un file di configurazione, sia da considerarsi un aspetto architetturale dell'applicazione in quanto non funzionale. Infatti tale

attività non riguarda una particolare funzionalità dell'applicazione stessa. In un'analisi e progettazione object oriented (cfr. capitolo 5), vengono affrontati argomenti che riguardano le funzionalità, tralasciando la definizione delle attività architetturali.

14.1.2 Internazionalizzazione (i18n)

Internazionalizzare un'applicazione, significa renderla capace di adattarsi automaticamente a vari linguaggi e zone. In inglese il termine **internationalization** è spesso abbreviato in **i18n**, poiché è un termine difficile da scrivere e pronunciare, e ci sono diciotto lettere tra la "i" iniziale e la "n" finale. Solitamente per gestire l'i18n di un'applicazione, si fa uso di diversi file di properties, ognuno dei quali contiene la rappresentazione in una lingua delle diverse stringhe visualizzate nell'applicazione. Vedremo come tali stringhe, invece di essere codificate staticamente all'interno dell'applicazione, saranno identificate da una stringa che rappresenta una **chiave**, e tradotta al volo da un meccanismo semiautomatico che individua il **valore** di tale chiave nel corretto file di properties.

Anche l'internazionalizzazione è da considerarsi un aspetto architetturale dell'applicazione in quanto non funzionale.

14.1.2.1 Classe `Locale` ed *internazionalizzazione*

Per gestire l'internazionalizzazione si deve utilizzare la classe `Locale`, che astrae il concetto di zona. Molte altre classi Java dipendono da `Locale`. Per esempio per rappresentare un numero decimale si utilizza la virgola come separatore tra il numero intero e le cifre decimali in Italia, mentre in America si utilizza il punto. La classe `NumberFormat` (package `java.text`) consente di interpretare e formattare un numero utilizzando un'istanza di `Locale`. Per esempio, con il seguente snippet:

```
NumberFormat nf = NumberFormat.getInstance(Locale.ITALIAN);
Number number = nf.parse("25,1");
```

abbiamo istanziato un oggetto `nf` di tipo `NumberFormat`, chiamando il metodo statico `getInstance()` della stessa classe, e passandogli il `Locale` italiano (costante statica `Locale.ITALIAN`). Tale oggetto quindi sarà capace di interpretare e formattare numeri secondo lo standard italiano. Infatti nell'oggetto `number` sarà registrato il valore `25.1`, visto che il metodo `parse()`, chiamato sull'oggetto `nf` interpreterà il numero con la virgola come numero decimale. Con il metodo `format()` invece, è possibile formattare un numero con uno specifico `Locale`:

```
  String s = nf.format(new Float(10.2));
  System.out.println(s);
```

che, grazie all'impostazione del `Locale` italiano, stamperà:

```
 10,2
```

Altri esempi di concetti che dipendono da `Locale` sono date, orari e valute. È possibile istanziare oggetti `Locale` utilizzando i suoi costruttori, per esempio possiamo istanziare il `Locale` inglese scrivendo:

```
  Locale englishLocale = new Locale("en");
```

dove la stringa `en` rappresenta una costante che rappresenta il linguaggio inglese e che andrà ad impostare la variabile d'istanza `language` della classe `Locale`.

È possibile anche istanziare un `Locale` specificando, oltre al linguaggio, una costante che rappresenta la nazione. Per esempio l'inglese si parla anche sia nel Regno Unito che negli Stati Uniti d'America. Quindi se volessimo istanziare un oggetto `Locale` statunitense, possiamo utilizzare il seguente costruttore:

```
  Locale usLocale = new Locale("en", "US");
```

dove il secondo parametro `US` è una costante che rappresenta la nazione e che andrà ad impostare la variabile d'istanza `country` della classe `Locale`. Nella documentazione ufficiale possiamo trovare le informazioni ed i link per conoscere tutte le costanti che ci servono. Esiste anche un terzo costruttore che usa come terzo parametro una costante che andrà ad impostare la variabile d'istanza `variant` della classe `Locale`. Con essa possiamo

specificare particolari costanti che ci consentiranno di specificare ulteriormente la zona. Per esempio si possono specificare zone dove si parlano particolari dialetti.

La classe `Locale` possiede inoltre alcuni metodi relativamente interessanti che restituiscono informazioni sulla zona. Ma probabilmente, più che interessarci dei metodi di `Locale`, ci dovremmo interessare all'utilizzo che ne fanno altre classi. Inoltre nella maggior parte dei casi non andremo ad istanziare `Locale`, bensì utilizzeremo le diverse costanti statiche di tipo `Locale` che individuano le zone considerate più facilmente utilizzabili. Esempi sono `Locale.US` (`Locale` statunitense), `Locale.FRANCE` (`Locale` francese) o `Locale.ITALY`.

A volte è comodo utilizzare il metodo `getDefault()`, che restituisce l'oggetto `Locale` prelevato dalle informazioni del sistema operativo. Per esempio EJE è un programma internazionalizzato. Con il seguente codice EJE riconosce il `Locale` da utilizzare per impostare la lingua (ulteriori dettagli saranno specificati nel prossimo paragrafo sul `ResourceBundle`):

```
static {
    //...
    Locale locale = null;
    String language =
EJE.properties.getProperty("eje.lang");
    locale = (language != null && !language.equals("")
              ? new Locale(language) :
Locale.getDefault());
    //...
}
```

In questo caso viene prima letto dal file di properties il valore di `eje.lang`. Poi viene controllato che sia stato valorizzato in qualche modo. Se esso è stato valorizzato, tramite un operatore ternario, viene assegnata alla variabile `locale` un'istanza di `Locale` inizializzata con il valore della variabile `language`. L'impostazione di questa variabile avviene mediante la scelta di un valore in un combo box (un menu a tendina) contenuto nella finestra di dialogo delle opzioni del programma (cfr. figura 14.1).

Figura 14.1 - Impostazione della variabile language dalla finestra di opzioni di EJE.

Se `language` non è mai stato impostato (condizione che dovrebbe presentarsi appena scaricato EJE), la variabile `locale` viene impostata al suo valore di default, ovvero quello relativo alla lingua del sistema operativo.

14.1.2.2 La classe `ResourceBundle`

Per completare il discorso sull'i18n dobbiamo parlare della classe `ResourceBundle`. Essa rappresenta un contenitore di risorse dipendente da `Locale`. Per esempio EJE "parlerà" inglese o italiano sfruttando un semplice meccanismo basato sul `ResourceBundle`. EJE utilizza al momento cinque file che si chiamano EJE.properties, EJE_en.properties, EJE_es.properties, EJE_de.properties ed EJE_it.properties che come al solito contengono coppie del tipo chiave-valore. I primi due file contengono tutte le stringhe personalizzabili in inglese (sono file identici), il terzo le stesse stringhe in spagnolo, il quarto in tedesco e il quinto in italiano. Il primo file EJE.properties, è considerato il file di default. Con l'ultimo esempio di codice del precedente paragrafo, abbiamo visto come viene scelto il `Locale`. Attualmente ci sono cinque possibili situazioni che si possono presentare quando si utilizza EJE:

1. il `Locale` è esplicitamente italiano perché la variabile `language` vale `"it"`;
2. il `Locale` è inglese perché la variabile `language` vale `"en"`;
3. il `Locale` è spagnolo perché la variabile `language` vale `"es"`;

4. il `Locale` è tedesco perché la variabile `language` vale `“de”;`
5. il `Locale` è quello di default (quindi potrebbe essere anche francese) perché la variabile `language` non è stata ancora valorizzata.

> **Per EJE il default è comunque inglese. Infatti i file** EJE.properties e EJE_en.properties **sono identici.**

Il codice che imposta l’oggetto `ResourceBundle` (di nome `resources`) da utilizzare è il seguente:

```
static {
    try {
        //...
        Locale locale = null;
        String language =
EJE.properties.getProperty("eje.lang");
        locale = (language != null && !language.equals("")
                  ? new Locale(language) :
Locale.getDefault());
        //...
        resources =
ResourceBundle.getBundle("resources.EJE", locale);
    } catch (MissingResourceException mre) {
        //...
    }
}
```

Dove la variabile `locale` è stata impostata come abbiamo visto nell’esempio precedente. Con il metodo `getBundle()` stiamo chiedendo di caricare un file che si chiama EJE nella directory `resources`, con il `locale` specificato. Se il `locale` specificato è italiano, viene caricato automaticamente il file EJE_it.properties; se il `locale` è inglese viene caricato il file EJE_en.properties, etc. Se il `locale` è per esempio francese, EJE prova a caricare un file di nome EJE_fr.properties che però non esiste, quindi viene caricato il file di default EJE.properties. Tutto sommato il codice scritto è minimo. Nel resto dell’applicazione, quando bisogna visualizzare una stringa internazionalizzata, viene semplicemente chiesto di farlo al `ResourceBundle`

mediante il metodo `getString()`. Per esempio nel seguente modo:

```
newMenuItem = new JMenuItem(resources.getString("file.new"),
  new ImageIcon("resources" + File.separator + "images" +
  File.separator + "new.png"));
```

la voce di menu (in inglese menu item) relativa al comando “nuovo file”, carica la sua stringa direttamente dal `ResourceBundle resources`. In questo caso carica il valore della proprietà che ha come chiave la stringa `file.new`, contenuta nel file di properties selezionato.

14.1.3 La classe `StringTokenizer`

`StringTokenizer` è una semplice classe che consente di separare i contenuti di una stringa in più parti, dette anche **token.** Un token è definito mediante la definizione di un **delimitatore del token**, ovvero una stringa che separa ogni token. Solitamente si utilizza per estrarre le parole da una stringa. Un token si può infatti definire come la sequenza massima di caratteri consecutivi che non sono delimitatori. I costruttori di `StringTokenizer` vogliono in input la stringa da analizzare per trovare i token. Il primo costruttore che vediamo non vuole altri parametri, ed assume come default i delimitatori “ `\t\n\r\f`” (si noti che il primo delimitatore è uno spazio). Quindi uno spazio, una tabulazione (`\t`), o l’andare a capo sono da considerarsi delimitatori (`\n` o `\r` o `\f`). Per esempio:

```
StringTokenizer st = new StringTokenizer("questo è un
test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

istanzia uno `StringTokenizer st`, che prende in input una stringa. Il metodo `hasMoreTokens()`, funziona in maniera molto simile al metodo `hasNext()` dell’interfaccia `Iterator` che abbiamo visto nel paragrafo 11.1.3. Quindi restituisce il valore booleano `true`, se nella stringa assegnata allo `StringTokenizer` esiste un token successivo. Mentre il metodo `next()`, fa avanzare un cursore sul token successivo della stringa (così come faceva il metodo `next()` di `Iterator` sugli elementi di una collezione). Quindi il

precedente snippet, stampa:

```
questo
è
un
test
```

Ma per uno `StringTokenizer` possiamo specificare anche delimitatori diversi, esplicitamente. Se utilizzassimo il seguente costruttore:

```
StringTokenizer st = new StringTokenizer("questo è un test",
";");
```

avremmo specificato il delimitatore come secondo parametro (`;`), e l'output sarebbe costituito da un unico token dato che non esiste quel delimitatore nella stringa specificata:

```
questo è un test
```

Esiste anche un terzo costruttore, che riceve in input un booleano. Se questo booleano vale `true`, lo `StringTokenizer` considererà token anche gli stessi delimitatori. Quindi l'output del seguente codice:

```
StringTokenizer st = new StringTokenizer("questo è un test",
"t", true);
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

sarà:

```
ques
t
o è un t
es
t
```

Mentre se utilizzassimo questo costruttore:

```
StringTokenizer st = new StringTokenizer("questo è un test",
"t", false);
```

l'output sarebbe:

```
ques
o è un
es
```

14.1.4 Formattazioni di output

Uno degli esempi più riusciti di utilizzo dei varargs è sicuramente il metodo `format()` della classe `java.util.Formatter`. Il metodo ha anche un overload che consente di specificare un `Locale` per effettuare eventuali formattazioni basate sull'internazionalizzazione. Seguono le dichiarazioni del metodo `format()`:

```
public Formatter format(String format, Object... args)
public Formatter format(Locale l, String format, Object...
args)
```

Il metodo `format()` serve per stampare un output, con la possibilità di formattare correttamente tutti gli input che gli vengono passati. Consideriamo il seguente esempio:

```
Formatter formatter = new Formatter(System.out);
formatter.format(Locale.ITALY, "e = %+10.4f", Math.E);
```

Esso stamperà il seguente output:

```
e = +2,7183
```

Ovvero, con la sintassi:

```
%+10.4f
```

abbiamo specificato, secondo la sintassi di formattazione definita dalla classe `Formatter`, il valore `double` della costante `E` della classe `Math`. In particolare, con il simbolo "`%`" avvertiamo il formatter che stiamo definendo un valore da formattare, in questo caso il primo (e l'unico) valore del parametro varargs, ovvero `Math.E`. Con il simbolo "`+`" invece, abbiamo specificato che l'output deve specificare il segno (positivo o negativo che sia). La `"f"` finale serve per specificare che l'output deve essere formattato come numero decimale (floating point). Il numero `"10"` che precede il "." fa in modo che l'output sia formattato in almeno dieci posizioni. Se l'output è costituito da meno di 10

posizioni come nel nostro caso, il valore viene emesso in dieci caratteri, allineato a destra, come è possibile notare dall'esempio. Infine, il numero "4" specificato dopo il "." indica che il valore deve essere emesso con quattro decimali (con eventuale arrotondamento). La classe offre la possibilità di formattare gli output con un numero enorme di varianti, che è difficile da memorizzare completamente. Fortunatamente la documentazione ufficiale della classe `Formatter`, si può e si deve consultare quando serve.

L'argomento sembra piuttosto complesso, ma esiste una categoria di persone che non sarebbe d'accordo con questa affermazione: i programmatori C/C++. Essi infatti hanno dovuto imparare ben presto le regole che governano la formattazione degli output. Infatti, la principale funzione di stampa del C, si chiama `printf()` e formatta gli output proprio con le regole a cui abbiamo appena accennato. Ecco che allora ci ritroviamo nella classe `PrintStream` un metodo chiamato proprio `printf()`. Questo ricalca la firma del metodo `format()` di `Formatter`, sfruttando come secondo parametro un varargs di `Object`. Infatti `printf()` non fa altro che chiamare a sua volta il metodo `format()`. Ma qual è l'oggetto di tipo `PrintStream` più famoso? Ma naturalmente `System.out`! Quindi sarà possibile utilizzare la seguente sintassi per stampare una data:

```
System.out.printf("Data %d", new Date());
```

14.1.5 Espressioni regolari

Esiste però un modo molto più potente di analizzare testo in Java. Infatti dalla versione 1.4, Java supporta le cosiddette **espressioni regolari** (in inglese **regular expressions).** Si tratta di una potente tecnica che esiste anche in altri ambienti (per esempio nei sistemi Unix e in linguaggi di programmazione come il Perl), per la ricerca e l'analisi del testo. Si basa su di un linguaggio sintetico e a volte criptico, composto appunto di espressioni, con una sintassi non ambigua, per individuare determinate aree di testo. Per esempio, l'espressione:

```
[aeiou]
```

consente di individuare una vocale. La sintassi è abbastanza vasta e quindi ci limiteremo a riportare solo quella fondamentale. Essenzialmente ci occorre

conoscere tre tipologie di espressioni: i **gruppi di caratteri (character classes)**, le **classi predefinite** e i **quantificatori (greedy quantifier).** Un esempio di **gruppi di caratteri** l'abbiamo appena visto:

```
[aeiou]
```

Che individua una tra le vocali. Per esempio l'espressione:

```
alunn[ao]
```

permette di individuare all'interno di un testo le occorrenze sia della parola "alunno" che della parola "alunna". L'operatore ^ è detto **operatore di negazione**, e per esempio l'espressione:

```
[^aeiou]
```

Individua un qualsiasi carattere escluse le vocali (quindi una consonante).

Con i gruppi di caratteri è anche possibile specificare range di caratteri con il separatore -. Per esempio con l'espressione:

```
[a-zA-Z]
```

è possibile individuare una qualsiasi lettera maiuscola o minuscola.

Le **classi predefinite** sono particolari classi che, con una sintassi abbreviata, consentono di definire alcune espressioni. Segue uno schema con tutte le classi predefinite e le relative descrizioni:

•	**Un qualsiasi carattere**
`\d`	Una cifra (equivalente a `[0-9]`)
`\D`	Una non-cifra (equivalente a `[^0-9]`)
`\s`	Un carattere spazio bianco (equivalente a `[\t\n\x0B\f\r]`)
`\S`	Un carattere non-spazio bianco (equivalente a `[^\s]`)
`\w`	Un carattere (equivalente a `[a-zA-Z_0-9]`)
`\W`	Un non carattere (equivalente a `[^\w]`)

Per quanto riguarda i **quantificatori**, questi servono a indicare la molteplicità di caratteri, parole, espressioni o gruppi di caratteri. In particolare:

- * significa zero o più occorrenze del pattern precedente
- + significa una o più occorrenze del pattern precedente
- ? significa zero o una occorrenza del pattern precedente

> **Stiamo parlando solo di una tipologia di quantificatori, i greedy quantifier, ma ne esistono altre con potenzialità notevoli.**

La libreria Java supporta le regular expression con tante classi: per esempio il metodo `replace()` della classe `String`, oppure la classe `Scanner` che introdurremo nel capitolo 18 riguardante l'Input-Ouput (la quale potrebbe tranquillamente sostituirsi anche alla classe `StringTokenizer`, ma usando regular expression). In realtà esiste il package `java.util.regex`, che definisce due semplici classi sulla quali si basa tutta la libreria che utilizza le espressioni regolari. Si tratta delle classi `Pattern` e `Matcher`. La classe `Pattern` serve proprio a definire le espressioni regolari mediante il suo metodo statico `compile()`. La classe `Matcher` invece definisce diversi metodi per la ricerca e l'analisi del testo, come i metodi `matches()`, `find()`, `start()`, `end()`, `replaceFirst()`, `replaceAll()`, etc. Per esempio EJE fa grande uso di regular expression per interpretare il codice digitato dallo sviluppatore. Il seguente codice:

```
Pattern p = Pattern.compile("\\bpackage\\b");
String content = EJEArea.this.getText();
Matcher m = p.matcher(content);
while (m.find()) {
    int start = m.start();
    int end = m.end();
    //...
```

permette di ricercare la posizione della parola `package` all'interno del testo eseguendo un ciclo su ogni occorrenza trovata. Infatti, i due `\b` che

circondano la parola `package`, fanno parte della sintassi delle espressioni regolari. Implicano che la stringa `package` debba essere separata da qualsiasi altro carattere. In pratica `\b` rappresenta quelli che erano i separatori di default dello `StringTokenizer`.

> **Si noti come sia stato necessario utilizzare due simboli di backslash invece che uno solo. Infatti il primo è necessario alla sintassi Java per interpretare il secondo come simbolo di backslash!**

Quindi la prima riga dello snippet ottiene un oggetto `Pattern p`, passando l'espressione regolare `\bpackage\b` al metodo statico `compile()`. Tale oggetto fa da wrapper Java per l'espressione regolare. Nella seconda riga la variabile `content` viene riempita con il testo digitato dall'utente in EJE. Nella terza viene creato un oggetto `Matcher` chiamando il metodo `matcher()` sull'oggetto `Pattern`, passando come input la stringa `content`. Anche il metodo `find()` chiamato sull'oggetto `Matcher m`, funziona in modo simile al metodo `hasNext()` di `Iterator`, ovvero cerca l'espressione regolare definita nell'oggetto `p`, all'interno della stringa `content`. Se non c'è coincidenza, allora il metodo ritorna il valore `false`. Se invece viene trovato un risultato, allora l'oggetto `m` sposta il cursore selezionando il risultato, e restituisce `true`. In tal caso entrerà nel ciclo e prenderà gli indici iniziali e finali della selezione nella stringa `content`, tramite la chiamata ai metodi `start()` e `end()`.

14.1.6 Date, orari e valute

Le classi `Date`, `Calendar`, `GregorianCalendar`, `TimeZone` e `SimpleTimeZone` consentono di gestire ogni tipologia di data e orario. È possibile anche gestire tali concetti sfruttando l'internazionalizzazione. Non sempre sarà semplice maneggiare date e orari e questo è dovuto proprio alla complessità dei concetti (a tal proposito basta leggere l'introduzione della classe `Date` nella documentazione ufficiale). La complessità è aumentata dal fatto che, oltre a queste classi, è molto probabile che serva utilizzarne altre

quali `DateFormat` e `SimpleDateFormat` (package `java.text`), le quali consentono la trasformazione da stringa a data e viceversa. Visto che il discorso è vasto e che nel prossimo paragrafo sarà introdotta la libreria destinata a rimodernare il modo in cui si gestiscono questi concetti, preferiamo solo accennare qualche esempio di utilizzo comune di queste classi. Per i dettagli si rimanda direttamente alla documentazione ufficiale. Tuttavia le classi spiegate in questo paragrafo sono comunque importanti visto che tutto il codice Java scritto sino all'avvento di Java 8 ne ha fatto uso. Nel paragrafo 14.2.5 vedremo anche come è possibile migrare dal vecchio al nuovo modo di gestire le date. Partiamo quindi direttamente con degli esempi.

Il seguente codice crea un oggetto `Date` (contenente la data corrente) e ne stampa il contenuto, formattandolo con un oggetto `SimpleDateFormat` dal pattern `dd-MM-yyyy`, ovvero due cifre per il giorno, due per il mese, e quattro per l'anno, con il separatore trattino – (la `d` minuscola indica il termine inglese "day" che significa giorno, la `M` maiuscola il termine "month" che significa mese, e la `y` minuscola corrisponde al termine "year" che significa anno):

```
Date date = new Date();
SimpleDateFormat df = new SimpleDateFormat("dd-MM-yyyy");
System.out.println(df.format(date));
```

I due seguenti frammenti di codice, invece, producono esattamente lo stesso output di formattazione di orario. Nel primo caso viene utilizzato un pattern personalizzato:

```
SimpleDateFormat formatter = new
SimpleDateFormat("HH:mm:ss");
String s = formatter.format(new Date());
```

La stringa `HH:mm:ss` sta ad indicare due cifre per l'ora, due per i minuti, e due per i secondi, separati dal simbolo `:` (la `H` maiuscola indica il termine inglese "hour" che significa ora, la `m` minuscola il termine "minute" che significa minuto, e la `s` minuscola corrisponde al termine "second" che significa secondo). Ecco l'output:

```
23:32:31
```

Che si sarebbe potuto ottenere anche con il seguente codice:

```
DateFormat sdf =
DateFormat.getTimeInstance(DateFormat.MEDIUM);
System.out.println(sdf.format(new Date()));
```

dove abbiamo utilizzato un'istanza di `DateFormat` passando la costante statica `DateFormat.MEDIUM` al metodo `getTimeInstance()`. Tale costante rappresenta proprio il pattern `HH:mm:ss`. Nel seguente codice invece raggiungiamo lo stesso risultato chiedendo di formattare allo stesso modo in maniera breve (`DateFormat.SHORT`) secondo lo stile italiano:

```
Locale locale = Locale.ITALY;
DateFormat formatter =
DateFormat.getTimeInstance(DateFormat.SHORT, locale);
String s = formatter.format(new Date());
System.out.println(s);
```

Nel prossimo esempio formattiamo due valute (o numeri) secondo gli standard americano ed italiano:

```
double number = 55667788.12345;
Locale localeUsa = Locale.US;
Locale localeIta = Locale.ITALY;

NumberFormat usaFormat =
NumberFormat.getInstance(localeUsa);
String usaNumber = usaFormat.format(number);
System.out.println(localeUsa.getDisplayCountry() + " " +
usaNumber);

NumberFormat itaFormat =
NumberFormat.getInstance(localeIta);
String itaNumber = itaFormat.format(number);
System.out.println(localeIta.getDisplayCountry() + " " +
itaNumber);
```

Output:

```
Stati Uniti 55,667,788.123
Italia 55.667.788,123
```

Si noti come l'oggetto `Locale` può essere utile per certe formattazioni. Nel

prossimo esempio utilizziamo una classe molto importante, non dal punto di vista della formattazione delle date, ma proprio per il calcolo: `Calendar` (e la sua sottoclasse `GregorianCalendar`). La classe `Date` non definisce alcuni metodi che sarebbero comodi. Tali metodi si possono trovare in `Calendar`. Per esempio, il metodo `get()`, utilizzando le costanti di `Calendar`, permette di recuperare i "pezzi" delle date:

```
public int getNumeroSettimana(Date date) {
    Calendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    int settimana = calendar.get(Calendar.WEEK_OF_YEAR);
    return settimana;
}
```

Esistono tante altre costanti statiche di `Calendar`: `DAY` (giorno), `MONTH` (mese), `YEAR` (anno), `DAY_OF_WEEK` (giorno della settimana), `DAY_OF_YEAR` (giorno dell'anno), `HOUR` (ora), etc. La sottoclasse `GregorianCalendar` possiede anche un costruttore, che prende come argomento un oggetto di tipo `TimeZone`, con il quale Java gestisce il concetto di fuso orario. Con questo costruttore è possibile ottenere orari relativi ad altre parti del mondo. Per esempio:

```
Calendar cal = new
GregorianCalendar(TimeZone.getTimeZone("America/Denver"));
int hourOfDay = cal.get(Calendar.HOUR_OF_DAY);
```

consente di vedere l'ora attuale a Denver negli Stati Uniti. Per ottenere tutti gli id validi per il metodo `getTimeZone()`, è possibile invocare il metodo statico `getAvailableIDs()` della stessa classe `TimeZone` che restituisce un array di stringhe.

14.2 La novità di Java 8: Date-Time API

Gestire date e orari in Java con le classi appena descritte non è mai stato semplice. Diversi casi particolari non sono per niente gestiti. Per esempio in alcune nazioni il passaggio dall'ora solare alla legale e viceversa è fatto più di una volta all'anno, e in altre ci sono degli anni dove il passaggio non viene proprio fatto. Altra difficoltà sorgono quando si utilizzano fusi orari, anni

bisestili o anni antecedenti al 1970. Infine lo standard su cui si basano le classi `Date` e `Calendar`, soffrono di piccole imprecisioni di calcolo temporale basandosi sui tempi della macchina su cui girano. I progettisti di Java 8 quindi, spinti dalla comunità Java, hanno deciso di implementare una nuova libreria nota come **Date-Time API.** Questa libreria contiene solo classi immutabili: una volta istanziati degli oggetti da esse, questi non possono più essere modificati. Inoltre si tratta di una libreria più chiara, efficiente, semplice da utilizzare ed estendibile. Questa API non si trova nel package `java.util`, bensì nel package `java.time` e nei suoi sottopackage. In particolare:

1. `java.time`: contiene le classi fondamentali della libreria, che rappresentano tempi, date, periodi, istanti, etc., basati sullo standard ISO-8601.
2. `java.time.chrono`: contiene le classi equivalenti a quelle di `java.time`, ma che seguono standard diversi da quelli di ISO-8601. È anche possibile creare nuovi calendari personalizzati.
3. `java.time.zone`: contiene le classi per gestire i fusi orari come `ZoneDateTime, ZoneOffset` e `ZoneId`.
4. `java.time.temporal`: package dedicato essenzialmente a chi deve estendere le funzionalità della libreria.
5. `java.time.format`: contiene le classi per formattare e interpretare le date e gli orari.

14.2.1 Standardizzazione dei metodi

Per renderne semplice l'utilizzo, la maggior parte dei metodi contenuti in questi package rispettano degli **standard per i nomi.** Per esempio in diverse classi esiste un metodo statico `from()` che istanzia un tipo della classe su cui si chiama il metodo, partendo dall'oggetto specificato come parametro del metodo. Per esempio il seguente frammento di codice:

```
LocalDate localDate = LocalDate.now();
YearMonth ym = YearMonth.from(localDate);
```

istanzia un oggetto `LocalDate` che rappresenta una data (come vedremo tra poco), tramite il metodo `now()`, che restituisce la data contestuale al momento in cui si esegue il programma. Poi il metodo statico `from()` della classe `YearMonth` restituisce un nuovo oggetto `YearMonth` a partire dall'oggetto `LocalDate`.

La seguente tabella riporta una serie di identificatori (o prefissi di identificatori) di metodi, utilizzati in molte classi della libreria Date-Time API. Si noti che non esistono metodi setter, visto che tutte le istanze della libreria sono immutabili. Ogni metodo che intende modificare un oggetto in essere, ne restituisce uno nuovo dello stesso tipo (un po' come avviene con i metodi della classe `String` o delle classi wrapper).

Nome / Prefisso	Static	Descrizione	Esempio
`now`	Si	Restituisce un'istanza della classe su cui è chiamato dell'istante in cui è invocato il metodo.	`LocalDate ora = LocalDate.now();`

`from`	Si	Restituisce un'istanza della classe su cui è chiamato partendo dal parametro in input che deve essere più completo (con più informazioni). Il metodo `from` quindi va usato quando ci serve qualche informazione rispetto all'oggetto passato in input.	`// localDate contiene l'informazione del`
			`// giorno in più rispetto a YearMonth`
			`YearMonth ym = YearMonth.from(localDate);`

`of`	Si	Restituisce un'istanza della classe su cui è chiamato dopo aver validato l'input.	`LocalDate fulvioBirth = LocalDate.of(1974, 9, 5);` `fulvioBirth = LocalDate.of(1974, Month.SEPTEMBER, 5);`
`with`	No	Si tratta di un prefisso al nome del metodo che può contenere altre parole. Restituisce una copia dell'oggetto passato in input con un elemento cambiato. È il metodo equivalente ad un metodo set per un oggetto immutabile.	`DateTimeFormatter =` `dateTimeFormatter.withLocale(Locale.FRENCH);`
`plus`	No	Restituisce una copia del parametro in input con l'aggiunta del tempo specificato.	`Instant traDieciMinuti =` `Instant.now().plus(10, ChronoUnit.MINUTES);`
`minus`	No	Restituisce una copia del parametro in input sottratto del tempo specificato.	`Instant dieciMinutiFa =` `Instant.now().minus(10, ChronoUnit.MINUTES);`
`get`	No	Si tratta di un prefisso al nome del metodo che può contenere altre parole. Ritorna una parte dei valori del primo parametro specificato.	`DayOfWeek dayOfWeek = date.getDayOfWeek();`
`is`	No	Si tratta di un prefisso al nome del metodo che può contenere altre parole. Ritorna lo stato dell'oggetto specificato.	`boolean isLeapYear = Year.of(2016).isLeap();`
`to`	No	Anche questo è un prefisso. Converte l'oggetto corrente in un altro tipo.	`int anni = lifeDuration.toYears();`
`at`	No	Restituisce un nuovo oggetto ottenuto aggiungendo le informazioni specificate all'oggetto corrente.	`ZonedDateTime zdt = ldt.atZone(losAngeles);`
`parse`	Si	Analizza la stringa in input per generare un'istanza della classe su cui è chiamato il metodo.	`LocalDate localDate = LocalDate.parse(data,` `DateTimeFormatter.ISO_LOCAL_DATE);`
`format`	No	Ritorna una stringa formattata usando il formatter specificato a partire dai valori nell'oggetto `Temporal` su cui è chiamato il metodo.	`LocalDate.now().format(` `DateTimeFormatter.ISO_LOCAL_DATE);`

14.2.2 Il package `java.time`

Questo package contiene definizioni di tempi, durate, date e istanti. A differenza delle classi storiche `Date` e `Calendar` questi concetti non sono basati sul tempo di sistema, bensì dal valore assoluto chiamato **timeline**, ovvero la **linea di tempo** misurata in nanosecondi che partono dal primo

istante del primo gennaio del 1970.

14.2.2.1 La classe `Instant`

Una delle classi più importanti del package `java.time` è la classe `Instant`. Essa rappresenta un istante nel tempo e internamente è rappresentato con i nanosecondi passati dalla mezzanotte del primo gennaio del 1970. Questa data simbolica è rappresentata con la costante statica di tipo `Instant` chiamata `EPOCH` definita all'interno della classe `Instant` stessa. Gli oggetti `Instant` che rappresentano istanti precedenti ad `EPOCH`, saranno identificati da un numero negativo, mentre quelli successivi ad `EPOCH` da un numero positivo. Le altre due costanti statiche definite dalla classe `Instant` sono `MIN` e `MAX` che astraggono rispettivamente il minimo `Instant` e il massimo `Instant` rappresentabili con questa classe. In particolare `MIN` rappresenta la data del primo gennaio dell'anno -1000000000 alle 00:00, mentre `MAX` rappresenta l'istante 31 dicembre dell'anno 1000000000 alle ore 23:59:5999999999, e questo intervallo di tempo dovrebbe risultare sufficiente per ogni scopo. `Istant` definisce metodi per sommare e sottrarre tempo di tipo "plus" e "minus". Per esempio:

```
Instant traDieciMinuti = Instant.now().plus(10,
ChronoUnit.MINUTES);
```

aggiunge dieci minuti all'istante attuale (`ChronoUnit` definisce le unità di misura temporali). Infatti il metodo `now()` restituisce l'istante attuale, e su quest'oggetto abbiamo invocato il metodo `plus()` che ha preso come primo parametro in input il valore intero `10` e come secondo parametro l'elemento `MINUTES` dell'enumerazione `ChronoUnit` del package `java.time.temporal`. Quest'ultimo è servito per specificare l'unità di misura temporale del primo parametro. Se avessimo usato `ChronoUnit.HOURS`, avremmo aggiunto 10 ore.

La classe `Instant`, definisce anche altri interessanti metodi come `isAfter()` che restituisce un booleano che indica se l'oggetto `Instant` su cui stiamo chiamando il metodo, è successivo all'oggetto `Instant` specificato in input. Per esempio, dopo aver eseguito:

```
Instant now = Instant.now();
boolean b = now.isAfter(Instant.EPOCH);
```

la variabile `b` varrà `true`. Equivalentemente esiste il metodo `isBefore()`.

Altro metodo degno di nota è il metodo `until()`. Per esempio per sapere quanti giorni avete vissuto sino a questo momento, dobbiamo istanziare un oggetto di tipo `Instant` che rappresenta l'istante della vostra nascita. Per esempio, sfruttando il metodo `parse()` nel seguente modo:

```
Instant nascita = Instant.parse("2004-04-14T07:00:00.00Z");
```

(più avanti parleremo di formattazioni e parsing di date), otteniamo questo oggetto che abbiamo chiamato `nascita`, e con la seguente riga di codice otterremo il numero di giorni che volevamo ottenere:

```
long giorniDallaNascita = nascita.until(Instant.now(),
ChronoUnit.DAYS);
```

14.2.2.2 Le classi `Duration` e `Period`

La classe `Duration`, come è facile immaginare, rappresenta un intervallo di tempo. A differenza della classe `Instant`, la classe `Duration` non è connessa alla linea temporale definita a partire da `Instant.EPOCH`, bensì, è basato sul tempo gestito dalla macchina dove gira il programma. Esso è calcolato in nanosecondi e non in ore, minuti o secondi. In generale un giorno dura sicuramente 24 ore, nessun ora solare, nessun fuso orario o altri fattori esterni. Questa classe va usata in generale per calcolare periodi da misurare con il tempo (ore, minuti, secondi, nanosecondi, etc.) e non con le date. Contiene il metodo statico `between()` che restituisce l'oggetto `Duration` compreso tra i due `Instant` specificati come parametro del metodo:

```
Duration duration = Duration.between(nascita,
Instant.now());
```

Inoltre definisce diversi metodi come `toDays()`, `toHours()`, `toSeconds()`, `toMillis()`, `toNanos()` ed altre decine di metodi di utilità per avere informazioni su periodi di tempo. La classe `Duration` si differenzia dalla classe `Period`, perché quest'ultima si basa su unità temporali quali anni, mesi e giorni. Definisce i metodi `getYears()`, `getMonths()` e `getDays()` per recuperare le informazioni. Per esempio, se vogliamo contare quanto tempo è passato dal nostro compleanno possiamo scrivere:

```
Period period = Period.between(mioCompleanno, oggi);
System.out.printf("Hai %s anni, %s mesi e %s giorni ",
  period.getYears(), period.getMonths(), period.getDays());
```

Un altro modo per calcolare un intervallo di tempo è offerto dal metodo `between()` definito nell'enumerazione `ChronoUnit` (cfr. paragrafo 14.2.4.1).

14.2.2.3 Le enumerazioni `DayOfWeek` e `Month`

Esistono delle semplici enumerazioni nel package `java.time`: `DayOfWeek` e `Month`, i cui elementi rappresentano semplicemente i giorni della settimana e i mesi dell'anno. I sette elementi di `DayOfWeek` si chiamano come prevedibile `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` e `SUNDAY`. Gli elementi di `Month` sono `JANUARY`, `FEBRUARY`, `MARCH`, `APRIL`, `MAY`, `JUNE`, `JULY`, `AUGUST`, `SEPTEMBER`, `OCTOBER`, `NOVEMBER` e `DECEMBER`. Associati a questi valori internamente ci sono dei valori interi che partono da `1` che si possono recuperare tramite il metodo `getValue()` definito per entrambe le enumerazioni. Per lo standard ISO-8601 la numerazione è la stessa che ci aspettiamo noi italiani con `DayOfWeek.MONDAY` e `Month.JANUARY` che hanno valore `1`. Inutile dire che la numerazione degli elementi delle due enumerazioni prosegue con `DayOfWeek.TUESDAY` e `Month.FEBRUARY` con valore 2 e così via, sino a `DayOfWeek.SUNDAY` con valore `7` e `Month.DECEMBER` con valore `12`.

Ci sono altri posti nel mondo dove per esempio il primo giorno della settimana è considerato domenica. Questo tipo di numerazione non è supportato dall'enumerazione `DayOfWeek`. Se comunque ci fosse un'esigenza simile, è possibile utilizzare la classe `WeekFields` che supporta la numerazione localizzata dei giorni della settimana, delle settimane del mese e delle settimane dell'anno.

14.2.2.4 Date classes: `LocalDate, YearMonth, MonthYear` *e* `Year`

Sono chiamate **Date Classes**, alcune classi che rappresentano date senza orario e fusi orari: `LocalDate, YearMonth, MonthDay`, e `Year`. La classe `LocalDate` rappresenta una data senza l'informazione del tempo. Per esempio con il metodo `of()`, possiamo ottenerne un'istanza in questo modo:

```
  LocalDate date = LocalDate.of(2004, Month.NOVEMBER, 12);
```

E poi è possibile per esempio recuperare il giorno della settimana nel seguente modo:

```
  DayOfWeek dayOfWeek = date.getDayOfWeek();
```

`YearMonth` invece rappresenta il mese di un anno. Per esempio il seguente frammento di codice:

```
  YearMonth yearMonth = YearMonth.now();
  YearMonth leapFebruary = YearMonth.of(2012, Month.FEBRUARY);
  System.out.println("Quest'anno dura: " +
  yearMonth.lengthOfYear()
    + " giorni");
  System.out.println("Il mese di febbraio 2012 è durato " +
    leapFebruary.lengthOfMonth() + " giorni");
```

stamperà:

```
Quest'anno dura: 365 giorni
Il mese di febbraio 2012 è durato 29 giorni
```

Infatti con la prima riga abbiamo istanziato un oggetto mediante il solito metodo `now()`. Con la seconda riga invece abbiamo istanziato un oggetto `YearMonth` che rappresenta il mese dell'anno 2012 che era bisestile *(leap* in inglese). Poi abbiamo stampato il numero di giorni dell'anno della variabile `yearMonth` sfruttando il metodo `lengthOfYear()` e la lunghezza del mese di febbraio 2012. La classe `MonthDay` rappresenta il giorno del mese, mentre `Year` rappresenta un anno. Sono rappresentazioni parziali delle date, per esempio con `MonthDay` è possibile rappresentare una certa data senza specificarne l'anno:

```
MonthDay md = MonthDay.of(Month.JULY, 29);
LocalDate ld = md.atYear(2012);
```

Con il metodo statico `of()` ne otteniamo un'istanza. Con il metodo `atYear()` aggiungiamo l'informazione dell'anno per ottenere un'istanza di `LocalDate`. Con `Year` è possibile rappresentare solo l'anno senza altri dettagli:

```
Year year = Year.of(2014);
```

Queste classi hanno tantissimi metodi di utilità classici come i metodi `of()`, `with()`, `now()`, `isBefore()`, `isAfter()`, etc.

14.2.2.5 Le classi `LocalTime` *e* `LocalDateTime`

La classe `LocalTime` invece è l'equivalente classe di `LocalDate`, ma che riporta solo informazioni sul tempo e non sulla data. Eseguendo in questo momento questa istruzione:

```
System.out.println("Ora: "+ LocalTime.now());
```

viene stampata l'ora esatta comprensiva di millisecondi:

```
Ora: 23:04:13.263
```

La classe `LocalDateTime`, altro non è che la classe che rappresenta sia le informazioni della data sia del tempo, e come è facile immaginare è una delle classi più importanti della libreria. È possibile pensare a questa come una combinazione tra `LocalDate` e `LocalTime`, tanto che tra i tanti metodi di tipo "of" ce n'è uno che prende proprio un `LocalDate` e un `LocalTime`:

```
LocalDateTime localDateTime =
LocalDateTime.of(LocalDate.now(), LocalTime.now());
```

Anche in questa classe troviamo tanti metodi per creare nuove istanze aggiungendo (metodi "plus") o sottraendo (metodi "minus") tempi e date.

14.2.2.6 Geolocalizzazione: le classi `ZonedDateTime`, `ZoneId` *e* `ZoneOffSet`

Nel package `java.time` esistono poi classi legate ai concetti di

localizzazione. Le prime due che vediamo sono `ZoneId` e `ZoneOffset`. In particolare `ZoneId` astrae il concetto di `TimeZone`, ovvero un'area geografica che condivide lo stesso orario. Di solito è univocamente individuata da una coppia del tipo "regione/città", per esempio "Europa/Berlin", "America/Detroit" oppure "Africa/Kinshasa". Per ottenere lo `ZoneId` di default per il nostro sistema basta chiamare il metodo statico `systemDefault()`:

```
ZoneId zoneId = ZoneId.systemDefault();
```

Per ottenere gli `ZoneId` disponibili esiste invece il metodo `getAvailableZoneIds()`:

```
Set<String> zoneIds = ZoneId.getAvailableZoneIds();
```

La classe `ZoneOffset` invece, astrae il concetto di fuso orario come differenza oraria rispetto all'orario ufficiale di Greenwich/UTC (UTC è l'orario ufficiale di Greenwich senza ora legale). Con il seguente codice capiamo qual è il fuso orario di Los Angeles:

```
LocalDateTime ldt = LocalDateTime.now();
ZoneId losAngeles = ZoneId.of("America/Los_Angeles");
ZonedDateTime zdt = ldt.atZone(losAngeles);
ZoneOffset offset = zdt.getOffset();
```

Per farlo abbiamo prima istanziato il `LocalDateTime` attuale. Poi con un metodo "of" abbiamo recuperato lo `ZoneId` di Los Angeles. Poi tramite il metodo `atZone()` abbiamo ottenuto una istanza di `ZonedDateTime`, che è la versione localizzata di `LocalDateTime`. Su di essa abbiamo chiamato il metodo `getOffSet()` che ci ha restituito l'oggetto `ZoneOffset`.

La classe `ZonedDateTime` ha in comune con `LocalDateTime` la maggior parte dei metodi. Esistono anche la classe `OffsetDateTime` che è equivalente a `ZonedDateTime` ma non contiene informazioni sullo `ZoneId`, e la classe `OffsetTime` che non contiene nemmeno le informazioni sulla data ma solo sull'offset del tempo.

Molte classi quindi definiscono il metodo `now()`, che restituisce una data basata sull'orologio del sistema su cui

viene eseguita l'istruzione e il default time zone. Se per esempio dobbiamo costruire un'applicazione internazionalizzata dove l'ora può cambiare a seconda della nazione dove ci si trova, allora potrebbe essere necessario utilizzare l'overload del metodo `now()` che prende in input un oggetto di tipo `Clock`. Si noti che questa classe è astratta e per istanziarla è necessario utilizzare i propri metodi statici (che ritorneranno oggetti istanziati da sottoclassi concrete di `Clock`).

14.2.3 Il package `java.time.format`

Questo package è essenzialmente definito da classi che consentono:

- il **parsing** di una stringa, ovvero l'interpretazione ed eventuale traduzione in un oggetto concreto di tipo data/orario;
- la **formattazione** di un oggetto di tipo data/orario, ovvero la traduzione in stringa leggibile di un oggetto che rappresenta una data/orario (azione complementare al parsing).

Una data si può infatti scrivere in tanti modi diversi, ed è naturale quindi che ci sia bisogno di un package dedicato. Una delle classi più importanti di questo package è la classe `DateTimeFormatter`. Essa definisce diversi costanti e metodi per formattare date e orari in formati standard. Per esempio con la seguente sintassi possiamo eseguire il parsing di una data sotto forma di stringa con formato `yyyy-MM-dd`:

```
String data = "1942-07-22";
LocalDate localDate = LocalDate.parse(data,
DateTimeFormatter.ISO_LOCAL_DATE);
```

Quindi l'oggetto predefinito `DateTimeFormatter.ISO_LOCAL_DATE` permette di interpretare stringhe con il formato `yyyy-MM-dd`. Esistono tanti altri formatter predefiniti che possono essere consultati direttamente dalla documentazione della classe `DateTimeFormatter` o all'indirizzo internet: http://docs.oracle.com/javase/9/docs/api/java/time/format/DateTimeFormatter
È possibile anche sfruttare quest'oggetto per formattare con questo formato

una stringa, come si può osservare nel seguente frammento di codice:

```
System.out.println(DateTimeFormatter.ISO_LOCAL_DATE.format(Loca
```

dove andiamo a stampare con il formato specificato la data odierna a partire da un oggetto `LocalDate`. Ecco l'output:

```
2017-12-25
```

Inoltre anche tutte le classi studiate di `java.time` (come appunto `LocalDate`) definiscono un metodo `format()` che prende in input un `DateTimeFormatter`. Quindi la seguente riga produce lo stesso output dell'esempio precedente:

```
System.out.println(LocalDate.now().format(DateTimeFormatter.ISO
```

Oltre ad utilizzare formatter predefiniti, è possibile anche utilizzare dei formatter che dipendono dalla localizzazione. Per esempio, con il seguente frammento di codice che usa anche un metodo "with":

```
DateTimeFormatter dateTimeFormatter =
  DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG);
dateTimeFormatter =
dateTimeFormatter.withLocale(Locale.FRENCH);
System.out.println(dateTimeFormatter.format(LocalDate.now()));
```

abbiamo ottenuto un `DateTimeFormatter` mediante la chiamata al metodo `ofLocalizedDate()` a cui abbiamo passato l'elemento `LONG` dell'enumerazione `FormatStyle`. Tale elemento implicherà una formattazione "estesa" della data localizzata. Non avendo specificato un `Locale`, è assunto che sarà preso il `Locale` della macchina attuale (presumibilmente il `Locale` italiano). Con la seconda riga in cui usiamo il metodo `withLocale()`, modifichiamo il `Locale` di default con il `Locale` francese, e infine stampiamo la data attuale formattata secondo il `Locale` francese. Infine è possibile anche creare pattern personalizzati per interpretare stringhe già in un certo formato, oppure formattare come stringhe delle date/orari a seconda della necessità. Per esempio, con il seguente codice:

```
DateTimeFormatter formatter =
  DateTimeFormatter.ofPattern("MMM/dd/yy hh:mm a");
String formattedDate =
formatter.format(LocalDateTime.now());
System.out.printf(formattedDate);
```

abbiamo deciso noi il pattern con il quale stampare la data. Si noti che per esempio specificando `MMM` formatteremo il mese con le prime tre lettere del nome, e non come numero. Mentre con `a` abbiamo aggiunto l'informazione AM/PM. Tutti i simboli e le lettere che vengono interpretati dai formatter sono descritti sempre nelle pagine di documentazione della classe `DateTimeFormatter` o all'indirizzo internet:

http://docs.oracle.com/javase/9/docs/api/java/time/format/DateTimeForma

> **È molto importante gestire eventuali eccezioni che possono essere lanciate quando si esegue il parsing o si formattano date e orari. In fondo le regole sono banali: se si formatta si deve gestire la `DateTimeException`, mentre se si esegue il parsing di una stringa si deve gestire una `DateTimeParseException`.**

14.2.4 Il package `java.time.temporal`

Questo package contiene classi, interfacce ed enumerazioni essenzialmente dedicate a gestire calcoli su date e orari, definire unità di tipo data/orario come anni, mesi etc., e differenti definizioni di settimane (per esempio quelle per cui il primo giorno della settimana è considerato la domenica e quelle per cui è considerato il lunedì). Questo package inoltre definisce l'interfaccia `Temporal`, che rappresenta un oggetto che ha a che fare con il tempo, e che è implementata da tutte le classi che rappresentano date e tempi come `Instant`, `LocalDate`, `LocalTime`, `LocalDateTime` e `ZonedDateTime`.

14.2.4.1 L'enumerazione `ChronoUnit`

Abbiamo già incontrato l'enumerazione `ChronoUnit` che appartiene al package `java.time.temporal`. Questa definisce con i suoi elementi le varie

unità di misure temporali utilizzate dalle classi della libreria Date-Time API, come `NANOS`, `MILLIS`, `MINUTES`, `DAYS`, `WEEKS CENTURIES`, `MILLENNIA`, `DECACDES` o `SECONDS` e così via. Questa enumerazione definisce anche un interessante metodo `between()` che restituisce il numero di giorni, ore, anni (a seconda dell'elemento dell'enumerazione usato) tra due momenti temporali. Per esempio, il seguente frammento di codice:

```
  ChronoUnit.DAYS.between(data1, data2);
```

restituisce il numero di giorni totali passati tra `data1` e `data2`.

14.2.4.2 Temporal Adjusters

I **temporal adjusters** sono classi ed interfacce che permettono di calcolare determinate date. La classe `TemporalAdjusters` (si noti la "s" finale) contiene decine di metodi di utilità, per esempio per cercare l'ultimo giorno del mese, l'ultima occorrenza di un certo giorno settimanale in un mese, il precedente venerdì e così via. In particolare forniscono metodi statici per la gestione di tipiche operazioni di calcolo di date e orari. I risultati di questi metodi possono essere passati a metodi `with()` per oggetti che rappresentano date e orari come `LocalDate` e `LocalTime`. Per esempio, nel seguente frammento di codice sfruttiamo la classe `TemporalAdjusters` per conoscere la prossima domenica:

```
  LocalDate oggi = LocalDate.now();
  System.out.printf("Prossima domenica :
    %s\n",
  oggi.with(TemporalAdjusters.nextOrSame(DayOfWeek.SUNDAY)));
```

con il metodo `nextOrSame()` abbiamo trovato la domenica successiva ad oggi (o oggi stesso se oggi fosse domenica). Il risultato di questo metodo viene passato in input al metodo `with()` che definisce l'oggetto `oggi` di tipo `LocalDate`, il quale restituisce proprio un oggetto di tipo `LocalDate` contenente la data desiderata. L'output di questo snippet è il seguente:

```
Prossima domenica :  2017-12-31
```

È anche possibile creare oggetti adjuster personalizzati creando classi che implementano l'interfaccia `TemporalAdjuster` (si noti che non stiamo

parlando della classe `TemporalAdjusters)` e ridefiniscono il metodo `adjustInto()`. Per esempio supponiamo di avere l'esigenza di voler pubblicare un nuovo post sul nostro blog ogni 4 giorni. Supponiamo anche di voler evitare di postare di domenica per qualche ragione. Allora potremmo creare un nostro adjuster con la seguente implementazione:

```
public class NextBlogPostAdjuster implements
TemporalAdjuster {
    public Temporal adjustInto(Temporal input) {
        LocalDate data = LocalDate.from(input);
        LocalDate result = data.plusDays(4);
        DayOfWeek giornoDellaSettimana =
result.getDayOfWeek();
        if (giornoDellaSettimana.equals(DayOfWeek.SUNDAY)) {
            result = result.plusDays(1);
        }
        return result;
    }
}
```

L'algoritmo è molto semplice: si crea un oggetto `data` di tipo `LocalDate` a partire dal parametro input del metodo, che essendo di tipo `Temporal` potrà essere un'istanza di una qualsiasi sua implementazione. Poi vengono aggiunti quattro giorni all'oggetto data, con il metodo `plusDays()`, ottenendo l'oggetto `result` di tipo `LocalDate`. Nel caso il risultato della computazione (la variabile `result)` coincide con una domenica, allora aggiungiamo un ulteriore giorno.

> **Ricordiamo che l'interfaccia `Temporal` è implementata da tutte le classi che hanno a che fare con il tempo come `ZonedDataTime`, `Instant`, `LocalDateTime`, `LocalDate`, `LocalTime` e così via.**

14.2.4.3 Temporal Queries

La classe `TemporalQueries` invece definisce un insieme di query predefinite (oggetti `TemporalQuery)` per ottenere informazioni quando l'applicazione non può risalire al tipo temporale che sta usando. Per esempio possiamo usare

il metodo `precision()` per ottenere la precisione per il tipo di dato che stiamo usando. Questo metodo restituisce la più piccola `ChronoUnit` del tipo di dato che viene interrogato. Per esempio, il seguente metodo:

```
public static void printPrecision(Temporal temporal) {
    TemporalQuery<?> query = TemporalQueries.precision();
    System.out.println(temporal.query(query));
}
```

una volta ricavato l'oggetto `TemporalQuery` tramite il metodo `precision()` chiamato su `TemporalQueries`, va a stamparne la precisione. Se fosse chiamato con le seguenti istruzioni:

```
printPrecision(LocalDate.now());
printPrecision(LocalTime.now());
```

stamperebbe:

```
Days
Nanos
```

Come con gli adjusters, anche in questo caso è possibile creare personalizzazioni di query, implementando l'interfaccia generica `TemporalQuery` e ridefinendo il metodo `queryFrom()`. Questo metodo restituirà il tipo parametrizzato di `TemporalQuery`. Per esempio la seguente classe implementa una query per capire se il giorno in input è un giorno del weekend:

```
public class IsWeekendQuery implements
TemporalQuery<Boolean> {
    public Boolean queryFrom(TemporalAccessor input) {
        LocalDate data = LocalDate.from(input);
        DayOfWeek giornoDellaSettimana =
data.getDayOfWeek();
        return
(giornoDellaSettimana.equals(DayOfWeek.SATURDAY) ||
                giornoDellaSettimana.equals(DayOfWeek.SUNDAY));

    }
}
```

Per prima cosa si noti che abbiamo parametrizzato con `Boolean` l'interfaccia

implementata. In questo modo il metodo `queryFrom()` ritornerà un `Boolean`. Poi abbiamo ritornato `true` o `false` (che per l'autoboxing sono compatibili con `Boolean`). Per usare la nostra query personalizzata possiamo scrivere:

```
LocalDate now = LocalDate.now();
boolean isWeekend = now.query(new IsWeekendQuery());
System.out.println(isWeekend);
```

che stamperà il risultato.

Le interfacce `TemporalQuery` e `TemporalAdjuster` sono interfacce funzionali, ovvero posseggono un solo metodo astratto. Questo significa che potremo utilizzare queste interfacce quando studieremo le espressioni lambda nel capitolo 16. Si noti anche che i metodi di `TemporalAdjuster` e quelli di `TemporalQueries` sono statici, e si adattano bene ad essere utilizzati con gli import statici.

14.2.5 Codice Legacy

In Java 8, con l'introduzione della libreria Date-Time API, sono stati introdotti dei metodi per interagire con il codice scritto in precedenza (codice legacy). Infatti per quasi vent'anni i programmatori Java hanno utilizzato le vecchie classi come `Date` e `Calendar` per utilizzare le date, e quindi sono state introdotte strategie per migrare ai tipi definiti dalla nuova libreria. In particolare con le vecchie classe temporali di `java.util` come `Date`, `Calendar`, `GregorianCalendar` e `TimeZone`, c'è la possibilità di migrare alle nuove implementazioni con uno sforzo tutto sommato modesto. In particolare sono state apportate le seguenti modifiche alle classi di `java.util`:

È stato introdotto il metodo `toInstant()` sia nella classe `Calendar`, che nella classe `Date`. Questo metodo ritorna un oggetto `Istant` a partire da un oggetto `Calendar` o da un oggetto `Date`. Per esempio sono valide le seguenti linee di codice:

```
Instant instant = new Date().toInstant();
Calendar calendar = Calendar.getInstance();
instant = calendar.toInstant();
```

Alla classe `Date` è stato aggiunto anche il metodo `from()` che prende in input un oggetto `Instant`, che crea un'oggetto `Date` a partire da un `Instant`.

Questo ci permetterà di creare un oggetto `Date`, a partire da un oggetto `Instant`. Per esempio è lecito scrivere:

```
Date date = Date.from(instant);
```

Inoltre nella classe `GregorianCalendar` sono state aggiunti i metodi `toZonedDateTime()`, che restituisce un oggetto `ZonedDateTime` a partire da un oggetto di tipo `GregorianCalendar`, e `from()`, che prende in input un oggetto `ZonedDateTime` per creare un `GregorianCalendar`. Per esempio è possibile scrivere:

```
GregorianCalendar gc =
GregorianCalendar.from(ZonedDateTime.now());
ZonedDateTime zdt = gc.toZonedDateTime();
```

Infine è stato aggiunto alla classe `TimeZone` il metodo `toZoneId()`, che restituisce un oggetto di tipo `ZoneId` a partire da un oggetto `TimeZone`. Segue un esempio:

```
TimeZone tz = TimeZone.getDefault();
ZoneId zi = tz.toZoneId();
```

In generale è possibile sostituire la classe `Date` con la classe `Instant`. La classe `GregorianCalendar` a seconda dei casi con le classi `ZonedDateTime`, `LocalTime` o `LocalDate`. E `TimeZone` con `ZoneId` o `ZoneOffset`. Per quanto riguarda le classi di formattazione, nonostante la classe `java.time.format.DateTimeFormatter` sia più completa, è ancora possibile utilizzare `DateFormat`, anche con le classi del package `java.time`, così come si faceva con le classi di `java.util`.

Riepilogo

La classe **Properties** estende la collection `Hashtable`, una mappa molto simile all'implementazione `HashMap`, e contiene coppie di tipo chiave-valore. La classe è capace di salvare in un file di testo (detto **file di properties)** tali coppie, e permette anche di leggere il contenuto di questo file. Essendo un file persistente, è possibile memorizzare nei file di properties dati che sopravvivranno alla chiusura della nostra applicazione e che potranno essere

riletti da essa alla prossima esecuzione. Quindi i file di properties sono molto utili per contenere informazione di configurazione di un'applicazione. Inoltre possono anche essere salvati in formato XML.

La classe **Preferences** può sostituire la classe `Properties` nelle sue funzionalità. La differenza essenziale consiste nel fatto che la classe `Preferences`, non usa un file per memorizzare le coppie chiave-valore, ma si basa su un meccanismo di persistenza che dipende dal sistema operativo su cui esegue e che possiamo tranquillamente ignorare. Questo significa che potremo modificare questi dati solo tramite l'applicazione stessa, e non andando a modificare a mano il file di properties.

L**'internazionalizzazione** (detta più brevemente **i18n)** ci consente di creare programmi che possono essere visualizzati in più lingue. Grazie alla classe **Locale**, alla classe **ResourceBundle** e a più file di properties, possiamo organizzare architetturalmente un'infrastruttura del programma in maniera standard. Si devono creare più file di properties, ognuno dei quali definisce una coppia chiave-valore per ogni stringa che l'utente dell'applicazione visualizzerà. Ogni file di properties avrà nomi simili che si differenzieranno in maniera standard per un suffisso che rappresenta la lingua (per esempio EJE_en.properties, EJE_it.properties e così via). Le stringhe dell'applicazione che l'utente visualizzerà quindi, non sono codificate staticamente all'interno del codice bensì sono prelevate dal corretto file di properties cercando la relativa chiave. La scelta del file di properties è definita dalla lingua specificata dal programma, grazie alla classe `ResourceBundle`. La classe **StringTokenizer**, ci permette di dividere i token di una stringa mediante la specifica di **separatori di token.**

La classe **Formatter** ci consente mediante il suo metodo **format()** di formattare stringhe con una sintassi standard che fa uso di caratteri jolly come `%s` o `%f` che vengono sostituiti al volo dagli elementi del varargs specificati.

Meno semplice è utilizzare le **espressioni regolari**, non tanto per le classi da utilizzare ma per la sintassi stessa delle espressioni, su cui esistono libri dedicati. Le regular expressions rappresentano uno strumento molto potente per l'analisi dei testi, e soprattutto rappresentano uno standard de facto dell'informatica da tanti anni.

Il nucleo di classi del package `java.util` costituito da **Date, Calendar, GregorianCalendar, TimeZone** e **SimpleTimeZone**, ha rappresentato il

modo standard per gestire date ed orari fino all'avvento di Java 8. La mancanza di flessibilità, la difficoltà di utilizzo e la mancanza di supporto a casi particolari hanno reso la libreria obsoleta e hanno favorito l'introduzione della libreria sostitutiva Date-Time API. Il supporto alla formattazione delle date era principalmente offerto invece dalle classi `DateFormat` e `SimpleDateFormat.`

Abbiamo visto come la nuova libreria **Date-Time API** sia composta da classi, interfacce ed enumerazioni che spesso sono calcolate a partire dalla "timeline", come `Instant.` In generale le classi sono molto semplici da utilizzare ed utilizzano una **standardizzazione dei nomi** dei metodi che ci permette di trovare in classi diverse metodi con nomi uguali (o prefissi di nomi uguali) e standard come "of", "now" e "with".

Le classi **Duration** e **Period** sono simili perché entrambe rappresentano intervalli di tempo. La seconda è più adatta però a gestire periodi di tempi più lunghi espressi quantomeno in giorni. Esistono enumerazioni molto coese a rappresentare astrazioni precise come **Month** o **DayOfWeek.**

Sono chiamate **Date Classes** alcune classi che rappresentano date senza orario e fusi orari: `LocalDate, YearMonth, MonthDay`, e `Year.` La classe **LocalDate** rappresenta una data senza l'informazione del tempo. Invece **YearMonth**, rappresenta un mese dell'anno, **MonthDay**, un giorno del mese, e **Year** rappresenta un anno. Ognuna di queste classi ha tanti metodi standard molto semplici per eseguire operazioni sulle date.

La classe **LocalTime** è l'equivalente classe di `LocalDate`, ma che riporta solo informazioni sul tempo e non sulla data. Ma la classe probabilmente più utilizzata della libreria è **LocalDateTime**, che rappresenta sia le informazioni della data sia del tempo. Può essere considerata come la sostituta della classe `java.util.Date.`

La **geolocalizzazione** si ottiene semplicemente usando classi geolocalizzate come **ZonedDateTime**, praticamente allo stesso modo di come si utilizzano quelle non geolocalizzate. Questa classe usufruisce del supporto delle classi **ZoneId** (che astrae il concetto di area geografica che condivide lo stesso orario) e **ZoneOffSet** (che astrae il concetto di fuso orario).

Il package `java.time.format` definisce classi che consentono il **parsing** di una stringa, e la **formattazione** di un oggetto di tipo data/orario in stringa. Per formattare ed analizzare le date esiste la classe `DateTimeFormatter`, che

da sola provvede ad ogni soluzione.

Il package **java.time.temporal** contiene classi, interfacce ed enumerazioni dedicate a gestire calcoli su date e orari, definire unità di tipo data/orario, e differenti definizioni di settimane. Definisce tra l'altro l'interfaccia **Temporal**, implementata da classi fondamentali come `Instant`, `LocalDateTime` e `ZonedDateTime`, e l'enumerazione **ChronoUnit** che definisce le unità di tempo.

Abbiamo visto come i **temporal adjusters** offrono strumenti per fare calcoli con le date, come per esempio ricercare il giorno del prossimo sabato (classe **TemporalAdjusters).** È possibile anche implementare l'interfaccia **TemporalAdjuster** per definire nuovi adjuster.

Le **temporal queries** permettono di ricavare informazioni quando l'applicazione non può risalire al tipo temporale che sta usando. Per esempio il metodo **precision()** della classe **TemporalQueries**, restituisce la minima unità di tempo `ChronoUnit` utilizzata dall'oggetto che si sta analizzando con il metodo **`query()`**. Con l'avvento di Date-Time API sono stati introdotti diversi **strumenti per migrare**, o quantomeno **fare interagire**, le classi storiche per la gestione delle date ed i nuovi meccanismi definiti dalla Date-Time API.

> **Esercizi e altro materiale didattico sono disponibili all'indirizzo** http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper implementare programmi configurabili mediante file di properties e preferences (unità 14.1)	☐	
Saper implementare programmi con l'internazionalizzazione (unità 14.1)	☐	
Saper utilizzare la classe `StringTokenizer` per suddividere stringhe (unità 14.1)	☐	

Saper utilizzare le basi delle espressioni regolari (unità 14.1)	☐	
Comprendere a grandi linee le classi `Date` e `Calendar` (unità 14.1)	☐	
Comprendere e saper utilizzare la nuova libreria Date-Time API (unità 14.2)	☐	

Note:

15

Gestione dei thread

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Saper definire multithreading e multitasking (unità 15.1).
- ✔ Comprendere la dimensione temporale introdotta dalla definizione dei thread in quanto oggetti (unità 15.2).
- ✔ Saper creare ed utilizzare thread tramite la classe `Thread` e l'interfaccia `Runnable` (unità 15.2).
- ✔ Definire che cos'è uno scheduler e i suoi comportamenti riguardo alle priorità dei thread (unità15.3).
- ✔ Saper utilizzare i modificatori `volatile` e `synchronized` (unità 15.3, 15.4).
- ✔ Far comunicare i thread (unità15.5).
- ✔ Conoscere il nucleo della libreria sulla concorrenza (unità15.6).

Questo capitolo è dedicato alla gestione della concorrenza dei processi (thread) nella programmazione Java. Non capita spesso di dover gestire i thread "a mano" nella programmazione giornaliera perché spesso in Java (per esempio nelle tecnologie Web implementate dalla Enterprise Edition) i thread sono gestiti automaticamente. Tuttavia riteniamo indispensabile la conoscenza di tali concetti per padroneggiare la programmazione Java e

comprendere molti concetti presenti nelle altre librerie. Per esempio, nella documentazione troveremo spesso la dicitura thread-safe (come nella classe `java.util.Hashtable` che sarà trattata nel capitolo relativo alle Collections). Questo capitolo quindi non è interamente dedicato alla libreria per gestire processi concorrenti. Cercheremo invece di capire i concetti alla base del meccanismo di gestione dei thread. Questo ci permetterà di comprendere appieno come funziona dietro le quinte il sistema che regola la concorrenza, consentendoci di risolvere problemi che altrimenti sarebbero difficili da risolvere senza questo sapere. Nella parte finale del capitolo, parleremo anche delle librerie che bisogna usare per semplificare il nostro lavoro, ma con maggiore consapevolezza dell'argomento.

Avvertiamo il lettore che non ha ancora familiarità con questi argomenti, che questo capitolo è particolarmente complesso e richiede impegno e concentrazione particolari. Chi troverà l'argomento troppo ostico potrà eventualmente tornare a rileggerlo in un secondo momento, quando si sentirà pronto. Per mitigare la complessità, cercheremo comunque di approcciare ai concetti in maniera graduale.

15.1 Introduzione ai thread

I thread rappresentano il mezzo mediante il quale Java fa eseguire un'applicazione da più Virtual Machine contemporaneamente, allo scopo di ottimizzare i tempi del runtime. Ovviamente si tratta di un'illusione: per ogni programma in esecuzione esiste un'unica JVM e in molti casi anche un'unica CPU. Ma la CPU può eseguire codice tramite più processi all'interno della gestione della JVM per dare l'impressione di avere più processori. Con processori multi-core il multi-threading darà il meglio di sé. L'esperienza ci ha insegnato che l'apprendimento di un concetto complesso come la gestione dei thread in Java richiede un approccio graduale, sia per quanto riguarda le definizioni sia per quanto riguarda le tecniche di utilizzo. Tale convincimento, oltre ad avere una natura empirica, nasce dall'esigenza di

dover definire un thread quale oggetto della classe `Thread`, cosa che può portare il lettore facilmente a confondersi. Didatticamente, inoltre, è spesso utile adoperare schemi grafici per aiutare nella comprensione. Ciò risulta meno fattibile quando si cerca di spiegare il comportamento di più thread al runtime, poiché uno schema statico potrebbe non risultare sufficiente.

15.1.1 Definizione provvisoria di thread

Quando eseguiamo un'applicazione Java vengono eseguite le istruzioni contenute in essa in maniera sequenziale, a partire dal codice del metodo `main()`. Spesso, soprattutto in fase di debug, allo scopo di simulare l'esecuzione dell'applicazione, il programmatore immagina un cursore che scorre sequenzialmente le istruzioni, magari simulando il suo movimento con un dito che punta sul monitor. Un tool di sviluppo che dispone di un debugger grafico, invece, evidenzia concretamente questo cursore. Possiamo per il momento identificare un thread proprio con questo **cursore immaginario.** In tal modo affronteremo le difficoltà dell'apprendimento in maniera graduale.

15.1.2 Cosa significa multithreading

L'idea di base è semplice: immaginiamo di eseguire un'applicazione Java. Il nostro cursore immaginario scorrerà ed eseguirà sequenzialmente le istruzioni partendo dal codice del metodo `main()`. Quindi, a runtime, esiste almeno un thread in esecuzione, ed il suo compito è eseguire il codice, seguendo il flusso definito dall'applicazione stessa.

Per **multithreading** si intende il processo che porterà un'applicazione a definire più thread, assegnando ad ognuno compiti da eseguire parallelamente. Il vantaggio che può portare un'applicazione multithreaded è relativo alle prestazioni della stessa. Infatti i thread possono dialogare allo scopo di spartirsi nella maniera ottimale l'utilizzo delle risorse del sistema. D'altronde, l'esecuzione parallela di più thread all'interno della stessa applicazione è vincolata all'architettura della macchina su cui eseguono. In altre parole, se la macchina monta un unico processore, in un determinato momento `x` può essere in esecuzione un unico thread. Ciò significa che un'applicazione non multithreaded che richiede in un determinato momento

alcuni input (da un utente, una rete, un database, etc.), per poter proseguire nell'esecuzione quando si trova in stato di attesa, non può eseguire nulla. Un'applicazione multithreaded, invece, potrebbe eseguire altro codice mediante un altro thread, avvertito dal thread che è in stato di attesa.

Solitamente il multithreading è una caratteristica dei sistemi operativi (per esempio Unix), piuttosto che dei linguaggi di programmazione. La tecnologia Java, tramite la Virtual Machine, ci offre uno strato d'astrazione per gestire il multithreading direttamente dal linguaggio. Altri linguaggi (come C/C++) solitamente sfruttano le complicate librerie del sistema operativo per gestire il multithreading, dove possibile. Infatti tali linguaggi, non essendo stati progettati per lo scopo, non supportano un meccanismo chiaro per gestire i thread.

Il multithreading non deve essere confuso con il multitasking. Possiamo definire "task" i processi pesanti, per esempio programmi come Word ed Excel. In un sistema operativo che supporta il multitasking è possibile eseguire più task contemporaneamente. La precedente affermazione può risultare scontata per molti lettori, ma negli anni '80 l'Home Computer utilizzava spesso sistemi chiaramente non multitasking. Ed anche negli ultimi anni (vedi il primo IPad) c'è stato qualche esempio. I task hanno comunque spazi di indirizzi separati e la comunicazione fra loro è limitata. I thread possono essere definiti processi leggeri, che condividono lo stesso spazio degli indirizzi e lo stesso processo pesante in cooperazione. I thread hanno quindi la caratteristica fondamentale di poter comunicare al fine di ottimizzare l'esecuzione dell'applicazione in cui sono definiti.

In Java i meccanismi della gestione dei thread risiedono essenzialmente:

1. nella classe `Thread` e nell'interfaccia `Runnable` (package `java.lang`);
2. nella classe `Object` (package `java.lang`);

3. nella JVM e nella keyword `synchronized`.

15.2 La classe `Thread` e la dimensione temporale

Come abbiamo precedentemente affermato, quando si avvia un'applicazione Java c'è almeno un thread in esecuzione, appositamente creato dalla JVM per eseguire il codice dell'applicazione. Nel seguente esempio introdurremo la classe `Thread` e vedremo che anche con un unico thread è possibile creare situazioni interessanti.

```
 1  public class ThreadExists {
 2     public static void main(String args[]) {
 3         Thread t = Thread.currentThread();
 4         t.setName("Thread principale");
 5         t.setPriority(10);
 6         System.out.println("Thread in esecuzione: " + t);
 7         try {
 8             for (int n = 5; n > 0; n--) {
 9                 System.out.println("" + n);
10                 t.sleep(1000);
11             }
12         }
13         catch (InterruptedException e) {
14             System.out.println("Thread interrotto");
15         }
16   }
17 }
```

Segue l'output dell'applicazione:

```
C:\TutorialJavaThread\Code>java ThreadExists
Thread in esecuzione: Thread[Thread principale,10,main]
5
4
3
2
1
```

15.2.1 Analisi di `ThreadExists`

Questa semplice classe produce un risultato tutt'altro che trascurabile: la gestione del tempo. La durata dell'esecuzione del programma è infatti quantificabile in circa cinque secondi. In questo testo è il primo esempio che gestisce in qualche modo la durata dell'applicazione stessa. Analizziamo il codice nei dettagli. Alla riga 3 viene chiamato il metodo statico `currentThread()` della classe `Thread`. Questo restituisce l'indirizzo dell'oggetto `Thread` che sta eseguendo l'istruzione e che viene assegnato al reference `t`. Questo passaggio è particolarmente delicato: abbiamo identificato un thread con il "cursore immaginario" che elabora il codice. Inoltre abbiamo anche ottenuto un reference a questo cursore. Una volta ottenuto un reference è possibile gestire il thread, controllando così l'esecuzione (temporale) dell'applicazione.

Notiamo la profonda differenza tra la classe `Thread` e tutte le altre classi della libreria standard. La classe `Thread` astrae un concetto che non solo è dinamico, ma addirittura rappresenta l'esecuzione stessa dell'applicazione. Il `currentThread` infatti non è l'"oggetto corrente", solitamente individuato dalla parola chiave `this`, bensì è l'oggetto che esegue l'oggetto corrente (ovvero il thread corrente). Potremmo affermare che un oggetto `Thread` si trova in un'altra dimensione rispetto agli altri oggetti: la dimensione temporale.

Continuiamo con l'analisi della classe `ThreadExists`. Alle righe 4 e 5 scopriamo che è possibile non solo assegnare un nome al thread, ma anche una priorità. La scala delle priorità dei thread in Java va dalla priorità minima 1 alla massima 10, e la priorità di default è 5.

Come vedremo più avanti, il concetto di priorità NON è la chiave per gestire i thread. Infatti, limitandoci alla sola gestione delle priorità, la nostra applicazione multithreaded potrebbe comportarsi in maniera differente su sistemi

diversi.

Alla riga 6 viene stampato l'oggetto `t` (ovvero `t.toString()`). Dall'output notiamo che vengono stampate informazioni sul nome e la priorità del thread, oltre che sulla sua appartenenza al gruppo dei thread denominato `main`. I thread infatti appartengono a un gruppo di thread astratto dalla classe `ThreadGroup`, ma non ci occuperemo di questo argomento in dettaglio perché poco interessante. Tra la riga 7 e la riga 12 viene dichiarato un blocco `try` contenente un ciclo `for` che esegue un conto alla rovescia da `5` a `1`. Tra una stampa di un numero e un'altra avviene una chiamata al metodo `sleep()` sull'oggetto `t`, a cui viene passato l'intero `1000`. In questo modo il thread che esegue il codice effettuerà una pausa di circa un secondo (1000 millisecondi) tra la stampa di un numero e il successivo.

Siccome l'implementazione di questo metodo si basa sul tempo fornito dal sistema operativo, non bisogna fare affidamento ciecamente sul fatto che la pausa sarà effettivamente di un secondo preciso.

Tra la riga 13 e la riga 15 viene definito il blocco `catch` per gestire una `InterruptedException` che il metodo `sleep()` dichiara nella sua clausola `throws`. Questa eccezione scatterebbe nel caso in cui il thread non riuscisse ad eseguire il suo codice perché interrotto da un altro thread tramite il metodo che si chiama per l'appunto `interrupt()`. Nel nostro esempio però, non vi è che un unico thread, e quindi la gestione dell'eccezione non ha molto senso. Nel prossimo paragrafo vedremo come creare altri thread sfruttando il thread principale.

15.2.2 L'interfaccia `Runnable` e la creazione dei thread

Per avere più thread basta istanziarne altri dalla classe `Thread`. Nel prossimo esempio noteremo che, quando si istanzia un oggetto `Thread`, bisogna passare al costruttore un'istanza di una classe che implementa l'interfaccia

`Runnable`. In questo modo il nuovo thread, quando sarà fatto partire (mediante la chiamata al metodo `start()`) eseguirà il codice del metodo `run()` dell'istanza associata. L'interfaccia `Runnable` quindi, richiede l'implementazione del solo metodo `run()` che definisce il comportamento di un thread, e l'avvio di un thread si ottiene con la chiamata del metodo `start()`. Dopo aver analizzato il prossimo esempio, le idee dovrebbero risultare più chiare.

```
1 public class ThreadCreation implements Runnable {
2     public ThreadCreation () {
3         Thread ct = Thread.currentThread();
4         ct.setName("Thread principale");
5         Thread t = new Thread(this, "Thread figlio");
6         System.out.println("Thread attuale: " + ct);
7         System.out.println("Thread creato: " + t);
8         t.start();
9         try {
10            Thread.sleep(3000);
11         }
12         catch (InterruptedException e) {
13             System.out.println("principale interrotto");
14         }
15         System.out.println("uscita Thread principale");
16    }
17   public void run() {
18     try {
19       for (int i = 5; i > 0; i--) {
20         System.out.println("" + i);
21         Thread.sleep(1000);
22       }
23     }
24     catch (InterruptedException e) {
25       System.out.println("Thread figlio interrotto");
26     }
27     System.out.println("uscita Thread figlio");
28  }
29  public static void main(String args[]) {
30    new ThreadCreation();
31  }
32 }
```

Segue l'output del precedente codice:

```
C:\TutorialJavaThread\Code>java ThreadCreation
Thread attuale: Thread[Thread principale,5,main]
Thread creato: Thread[Thread figlio,5,main]
5
4
3
uscita Thread principale
2
1
uscita Thread figlio
```

15.2.3 Analisi di `ThreadCreation`

Nel precedente esempio, oltre al thread principale, ne è stato istanziato un secondo. La durata dell'esecuzione del programma è anche in questo caso quantificabile in circa cinque secondi.

Analizziamo il codice nei dettagli. L'applicazione al runtime viene eseguita a partire dal metodo `main()` alla riga 29. Alla riga 30 viene istanziato un oggetto della classe `ThreadCreation;` poi il nostro cursore immaginario si sposta ad eseguire il costruttore dell'oggetto appena creato, alla riga 3. Qui il cursore immaginario (il thread corrente) ottiene un reference `ct`. Notiamo che "`ct` esegue `this`". A `ct` viene poi assegnato il nome "thread principale". Alla riga 5 viene finalmente istanziato un altro thread, dal thread corrente `ct` che sta eseguendo la riga 5 (figura 1 dell'applet e figura 15.2 di questo manuale; in figura 15.1 è possibile vedere la notazione usata negli schemi).

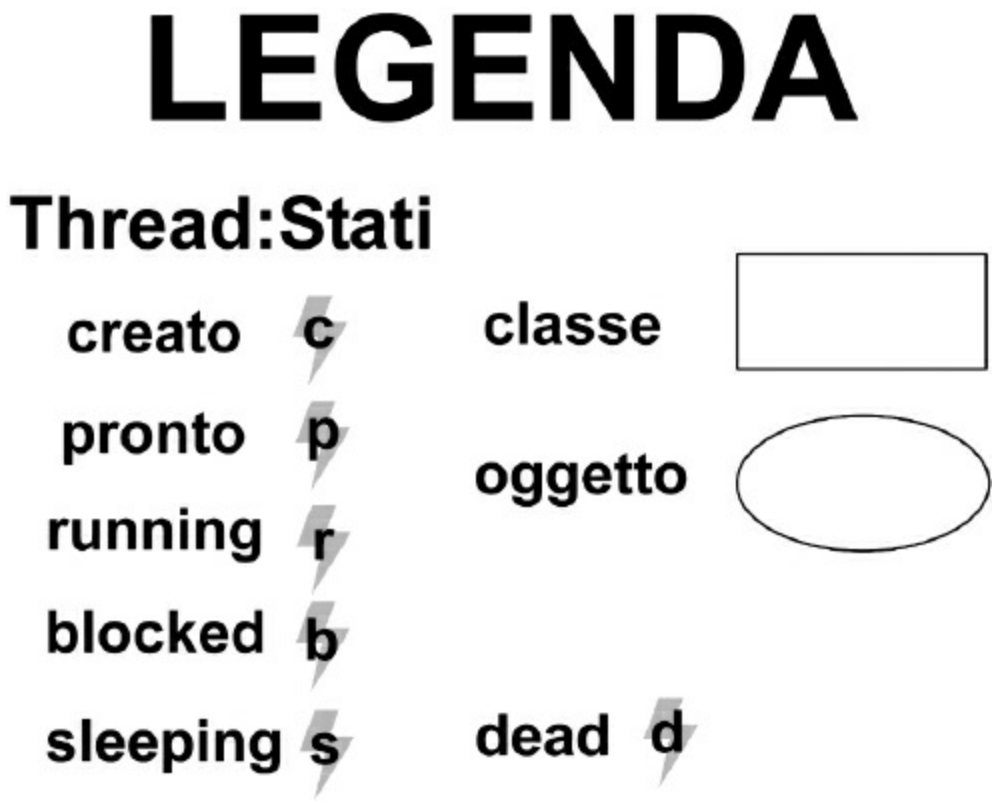


Figura 15.1 - Elementi della notazione degli schemi.

È utilizzato un costruttore che riceve in input due parametri. Il primo (`this`) è un oggetto `Runnable` (ovvero un'istanza di una classe che implementa l'interfaccia `Runnable);` il secondo è il nome del thread. L'oggetto `Runnable` contiene il metodo `run()` che diventa l'obiettivo dell'esecuzione del thread `t`. Quindi, mentre per il thread principale l'obiettivo dell'esecuzione è scontato, per i thread che sono istanziati bisogna specificarlo passando al costruttore un oggetto `Runnable`. Alle righe 6 e 7 vengono stampati messaggi descrittivi dei due thread. Alla riga 8 viene finalmente fatto partire il thread `t` mediante il metodo `start()`.

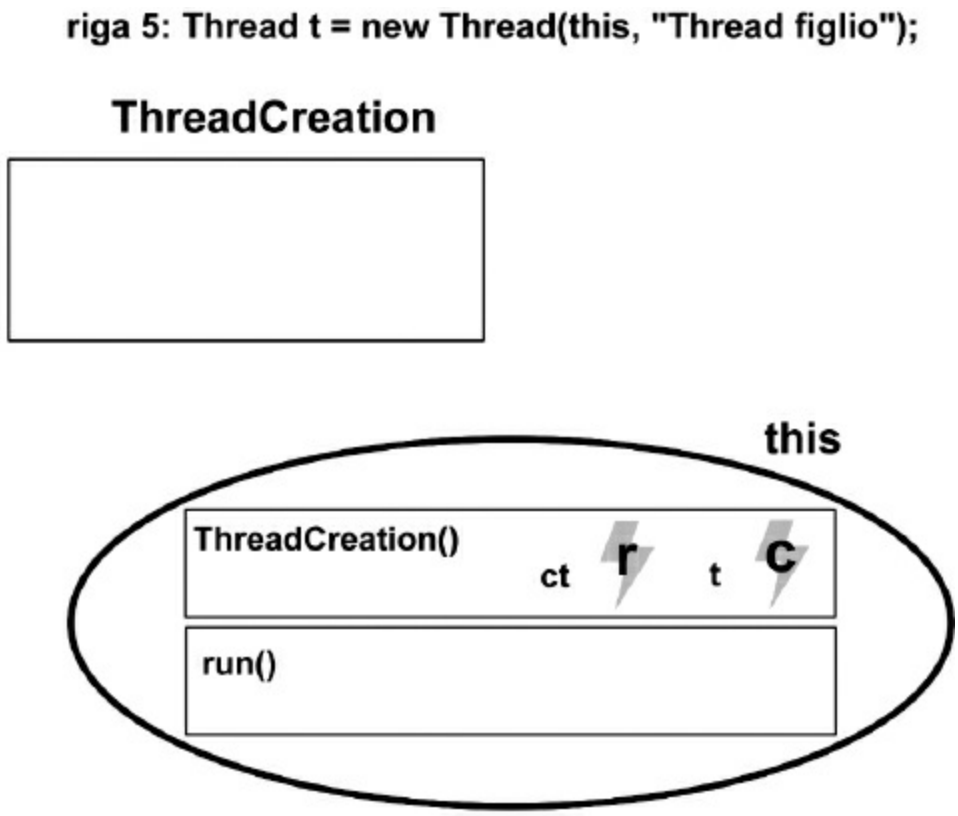


Figura 15.2 - Istanza del Thread.

La chiamata al metodo `start()` per eseguire il metodo `run()` fa sì che il thread `t` vada prima o poi ad eseguire il metodo `run()`. Invece, una eventuale chiamata del metodo `run()` non produrrebbe altro che una normale esecuzione dello stesso da parte del thread principale `ct` e non ci sarebbe multithreading.

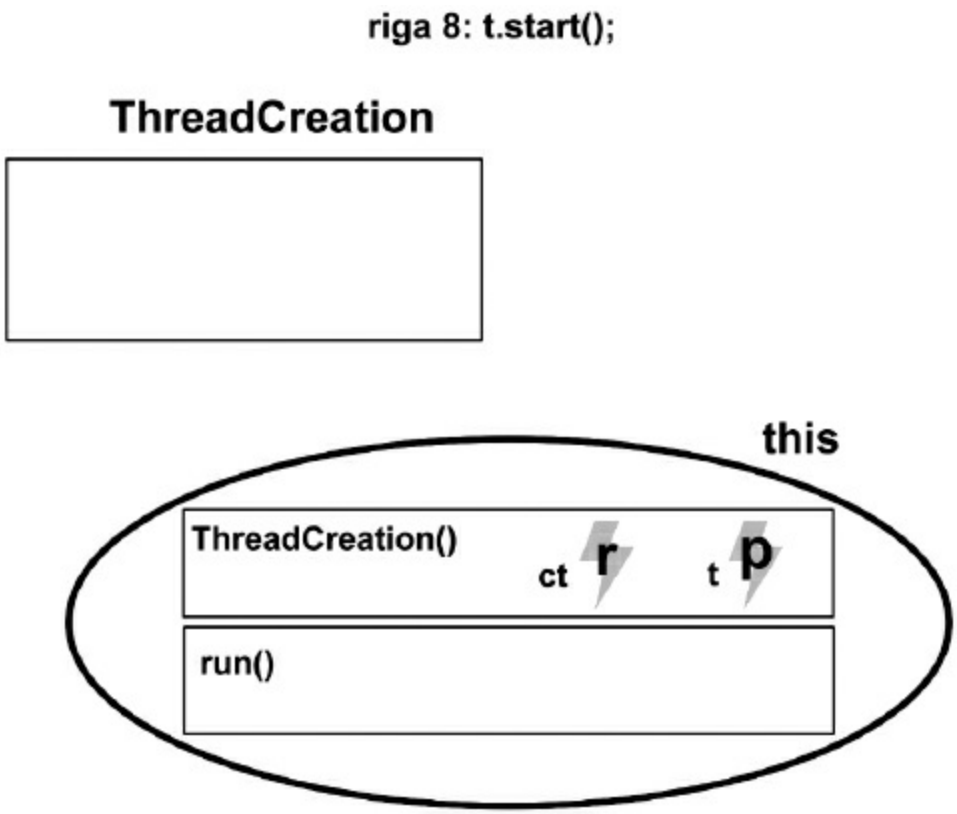


Figura 15.3 - Chiamata al metodo start().

La partenza (tramite l'invocazione del metodo `start()`) di un thread **non** implica che il thread inizi immediatamente ad eseguire il codice, ma solo che è stato reso eleggibile per l'esecuzione (figura 15.3). Quindi, il thread `ct`, dopo aver istanziato e reso eleggibile per l'esecuzione il thread `t`, continua nella sua esecuzione fino a quando, giunto ad eseguire la riga 10, incontra il metodo `sleep()` che lo ferma per 3 secondi.

> **Si noti come il metodo `sleep()` sia statico. Infatti viene mandato a dormire il thread che esegue il metodo.**

A questo punto il processore è liberato dall'esecuzione del thread `ct` e viene utilizzato dal thread `t`, che finalmente può eseguire il metodo `run()`. Ecco che allora il thread `t` esegue un ciclo, che, come nell'esempio precedente, realizza un conto alla rovescia, facendo pause da un secondo. Esaminando l'output verifichiamo come il codice faccia sì che il thread `t` stampi il `5`

(figura 15.4), faccia una pausa di un secondo (figura 15.5), stampi `4`, si metta in pausa per un secondo, stampi `3`, effettui un'altra pausa di un secondo.

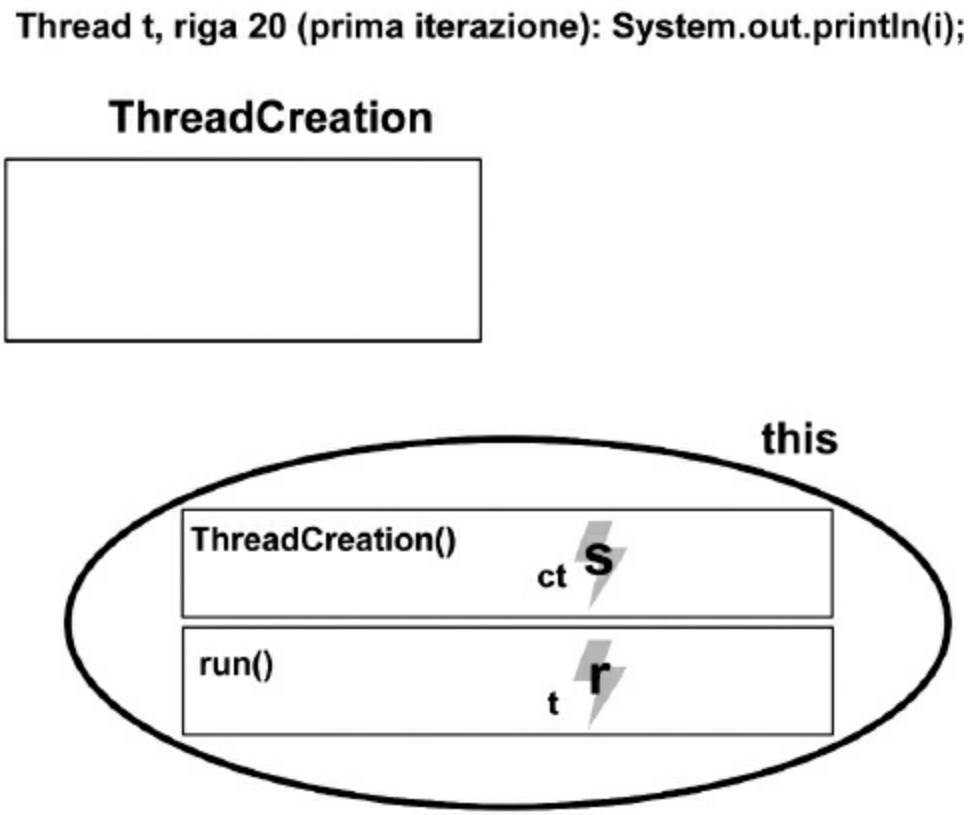


Figura 15.4 - Prima iterazione, stampa 5.

Poi si risveglia il thread `ct`, che stampa la frase "uscita thread principale" (figura 15.6) e "muore".

Quasi contemporaneamente viene stampato `2` (figura 15.7), cui segue una pausa di un secondo, la stampa di `1`, ancora una pausa di un secondo e la stampa di "uscita thread figlio".

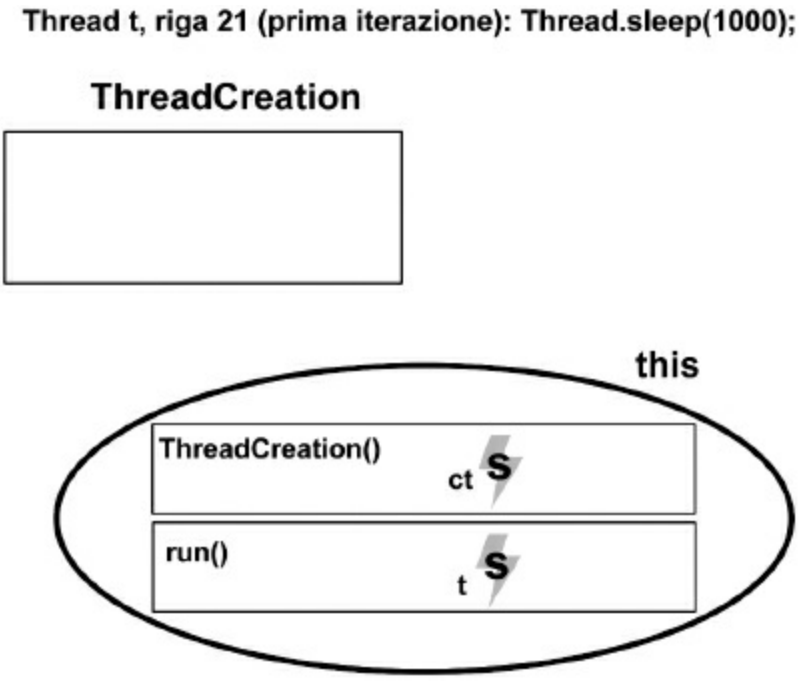


Figura 15.5 - Prima iterazione, tutti i thread in pausa.

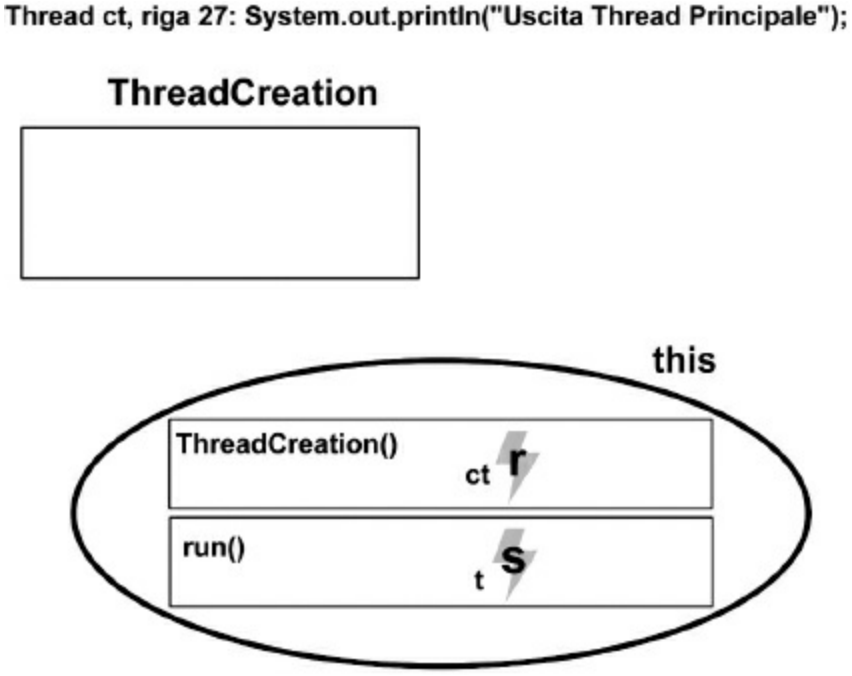


Figura 15.6 - Uscita thread principale.

Nell’esempio, quindi, l’applicazione ha un ciclo di vita superiore a quello del thread principale, grazie al thread creato.

Un altro metodo simile al metodo `sleep()` è il metodo `join()`. A differenza di `sleep()` però, `join()` non è un metodo statico. Va chiamato su una particolare istanza. In particolare, supponiamo che un certo thread stia eseguendo una certa parte di codice. Se trova sulla sua strada un'espressione del tipo:

```
t.join()
```

dove `t` è un altro thread in esecuzione, il thread corrente si metterà in pausa sino a quando il thread `t` non avrà completato il suo compito.

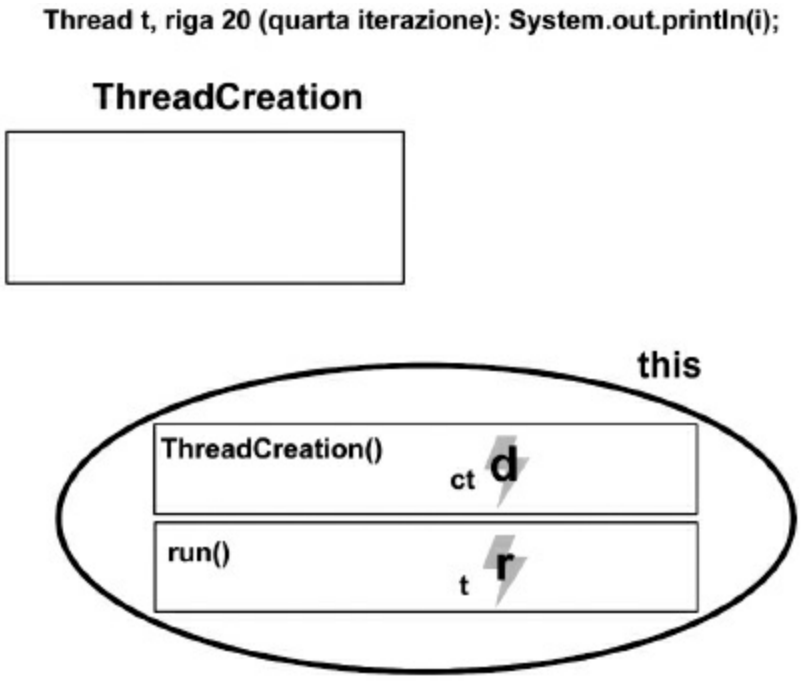


Figura 15.7 - Quarta iterazione.

15.2.4 La classe `Thread` e la creazione dei thread

Abbiamo appena visto come un thread creato deve eseguire codice di un oggetto istanziato da una classe che implementa l'interfaccia `Runnable` (l'oggetto `Runnable`). Ma la classe `Thread` stessa implementa l'interfaccia `Runnable`, fornendo un'implementazione vuota del metodo `run()`. È quindi possibile fare ese-guire ad un thread il metodo `run()` definito all'interno dello

stesso oggetto thread. Per esempio:

```
1  public class CounterThread extends Thread {
2     public void run() {
3         for (int i = 0; i<10; ++i)
4             System.out.println(i);
5     }
6  }
```

Si può istanziare un thread senza specificare l'oggetto `Runnable` al costruttore, e farlo partire con il solito metodo `start()`:

```
1 CounterThread thread = new CounterThread();
2 thread.start();
```

È possibile (ma non consigliato) creare un thread che utilizza un `CounterThread` come oggetto `Runnable`:

```
Thread t = new Thread(new CounterThread());
 t.start();
```

Sicuramente la strategia di fare eseguire il metodo `run()` all'interno dell'oggetto thread stesso è più semplice rispetto a quella vista nel paragrafo precedente. Tuttavia ci sono almeno tre buone ragioni per preferire il passaggio di un oggetto `Runnable`:

1. in Java una classe non può estendere più di una classe alla volta. Quindi implementare l'interfaccia `Runnable`, piuttosto che estendere `Thread`, consentirà di utilizzare l'estensione per un'altra classe.
2. Solitamente un oggetto della classe `Thread` non dovrebbe possedere variabili d'istanza private che rappresentano i dati da gestire. Quindi il metodo `run()` nella sottoclasse di `Thread` non potrà accedere, perlomeno in modo pulito, a tali dati.
3. Dal punto di vista della programmazione object-oriented, una sottoclasse di `Thread` che definisce il metodo `run()` combina due funzionalità poco relazionate tra loro: il supporto del multithreading ereditato dalla classe `Thread` e l'ambiente esecutivo fornito dal metodo `run()`. Quindi in questo caso l'oggetto creato è un thread

associato con sé stesso, e questa non si può considerare una soluzione propriamente object oriented.

15.3 Priorità, scheduler e sistemi operativi

Abbiamo visto come il metodo `start()` chiamato su un thread non implichi che questo inizi immediatamente ad eseguire il suo codice (contenuto nel metodo `run()` dell'oggetto associato). La JVM definisce un **thread scheduler** che si occuperà di decidere in ogni momento quale thread deve trovarsi in esecuzione. Il problema è che la JVM stessa è un software in funzione su un determinato sistema operativo, e la sua implementazione dipende dal sistema. Quando si gestisce il multithreading ciò può apparire evidente, come nel prossimo esempio. Infatti lo scheduler della JVM deve comunque rispettare la filosofia dello scheduler del sistema operativo, e questa può cambiare notevolmente tra sistema e sistema. Prendiamo in considerazione solo due sistemi: Unix (Solaris) e Windows (qualsiasi versione). Il seguente esempio produce sui due sistemi output completamente diversi:

```
1  public class Clicker implements Runnable {
2      private long click = 0L;
3      private Thread t;
4      private volatile boolean running = true;
5      public Clicker(int p) {
6          t = new Thread(this);
7          t.setPriority(p);
8      }
9      public long getClick() {
10         return click;
11     }
12     public void run() {
13         while (running) {
14             click++;
15         }
16     }
17     public void stopThread() {
18         running = false;
19     }
20     public void startThread() {
21         t.start();
```

```
 22     }
 23 }
 24 public class ThreadRace {
 25     public static void main(String args[]) {
 26         Thread.currentThread().setPriority(Thread.MAX_PRIORI

 27         Clicker hi = new Clicker(Thread.NORM_PRIORITY +
 2);
 28         Clicker lo = new Clicker(Thread.NORM_PRIORITY -
 2);
 29         lo.startThread();
 30         hi.startThread();
 31         try {
 32             Thread.sleep(10000);
 33          }
 34          catch (Exception e) {}
 35          lo.stopThread();
 36          hi.stopThread();
 37          System.out.println(lo.getClick()+" vs." +
 hi.getClick());
 38    }
 39  }
```

Segue l'output su Sun Solaris 8:

```
solaris% java ThreadRace
0 vs. 1963283920
```

Vari output su Windows 8:

```
C:\TutorialJavaThread\Code>java ThreadRace
4880762242 vs. 4872062039
C:\TutorialJavaThread\Code>java ThreadRace
4850917490 vs. 4824754791
C:\TutorialJavaThread\Code>java ThreadRace
4915745356 vs. 4920787825
C:\TutorialJavaThread\Code>java ThreadRace
 4942114184 vs. 4947401095
```

15.3.1 Analisi di `ThreadRace`

Nell'esempio si fa uso di un modificatore sino ad ora non

illustrato, `volatile`, **di cui parleremo nel paragrafo 15.3.4.**

Il precedente esempio è composto da due classi: `ThreadRace` e `Clicker`. `ThreadRace` contiene il metodo `main()` e rappresenta quindi la classe principale. Immaginiamo di eseguire l'applicazione. Alla riga 26 viene assegnata al thread corrente la priorità massima tramite la costante statica intera della classe `Thread MAX_PRIORITY`, che vale 10. Alle righe 27 e 28 vengono istanziati due oggetti dalla classe `Clicker, hi` e `lo`, ai cui costruttori vengono passati i valori interi 7 e 3. Gli oggetti `hi` e `lo`, tramite costruttori, creano due thread associati ai propri metodi `run()` rispettivamente con priorità 7 e priorità 3. Il thread principale continua nella sua esecuzione chiamando su entrambi gli oggetti `hi` e `lo` il metodo `startThread()`, che rende eleggibili per l'esecuzione i due thread a priorità 7 e 3 nei rispettivi oggetti (figura 15.8).

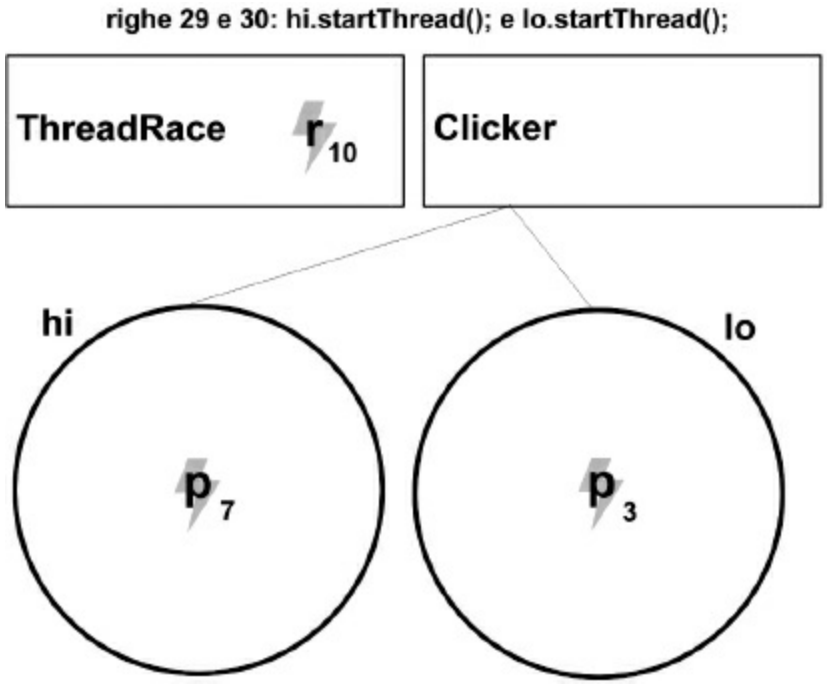


Figura 15.8 - Start dei thread.

Alla riga 32, il thread principale a priorità 10 va a dormire per una decina di secondi, lasciando disponibile la CPU agli altri due thread (figura 15.9).

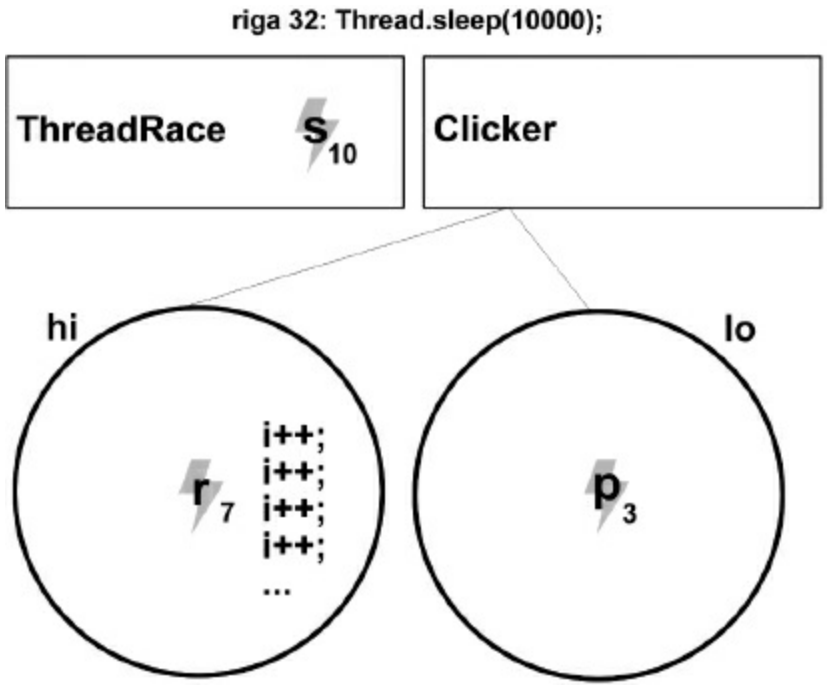


Figura 15.9 - Il thread principale si addormenta.

È in questo momento dell'esecuzione dell'applicazione che lo scheduler avrà un comportamento dipendente dalla piattaforma.

15.3.2 Comportamento Windows (e la maggior parte dei sistemi Linux)

Il comportamento su Windows e la maggior parte dei sistemi Linux viene chiamato **time slicing** o **round-robin scheduling.** Un thread può trovarsi in esecuzione solo per un certo periodo di tempo, poi deve lasciare ad altri thread la possibilità di essere eseguiti. Ecco che allora l'output di Windows evidenzia che entrambi i thread hanno avuto la possibilità di eseguire codice. Il thread a priorità 7 ha avuto a disposizione per molto più tempo il processore, rispetto al thread a priorità 3. Tale comportamento è però non deterministico e quindi l'output prodotto cambierà anche radicalmente ad ogni esecuzione dell'applicazione.

15.3.3 Comportamento Unix

Il comportamento sui sistemi UNIX viene detto **preemptive scheduling.** Un thread in esecuzione può uscire da questo stato solo nelle seguenti situazioni:

1. viene chiamato un metodo di scheduling come `wait()` o `suspend()`;
2. viene chiamato un metodo di blocking, come quelli di I/O;
3. può essere "buttato fuori" dalla CPU da un altro thread a priorità più alta che diviene eleggibile per l'esecuzione;
4. termina la sua esecuzione (ovvero termina il suo metodo `run()`).

Quindi il thread a priorità 7 ha occupato la CPU per tutti i 10 secondi nei quali il thread principale (a priorità 10) è in pausa. Quando l'altro thread si risveglia, rioccupa di forza la CPU.

Su entrambi i sistemi l'applicazione continua con il thread principale che, chiamando il metodo `stopThread()` (righe 35 e 36) su entrambi gli oggetti `hi` e `lo`, imposta le variabili `running` a `false` (figure 15.10 e 15.11).

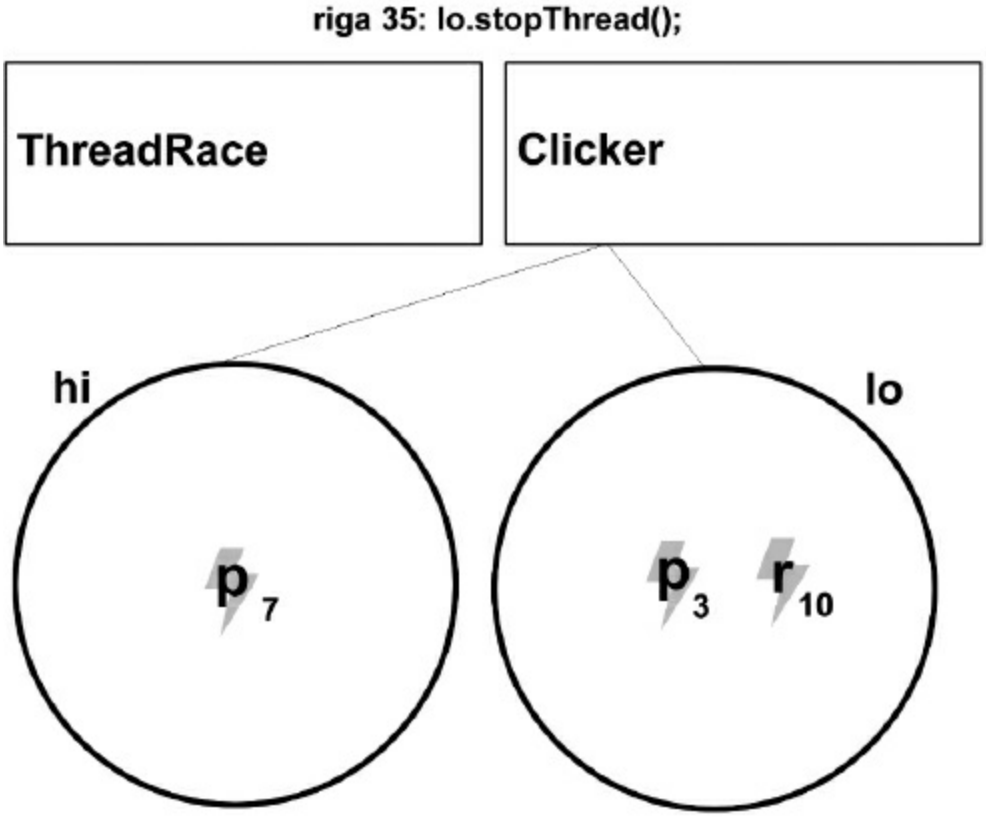


Figura 15.10 - Stop del thread lo.

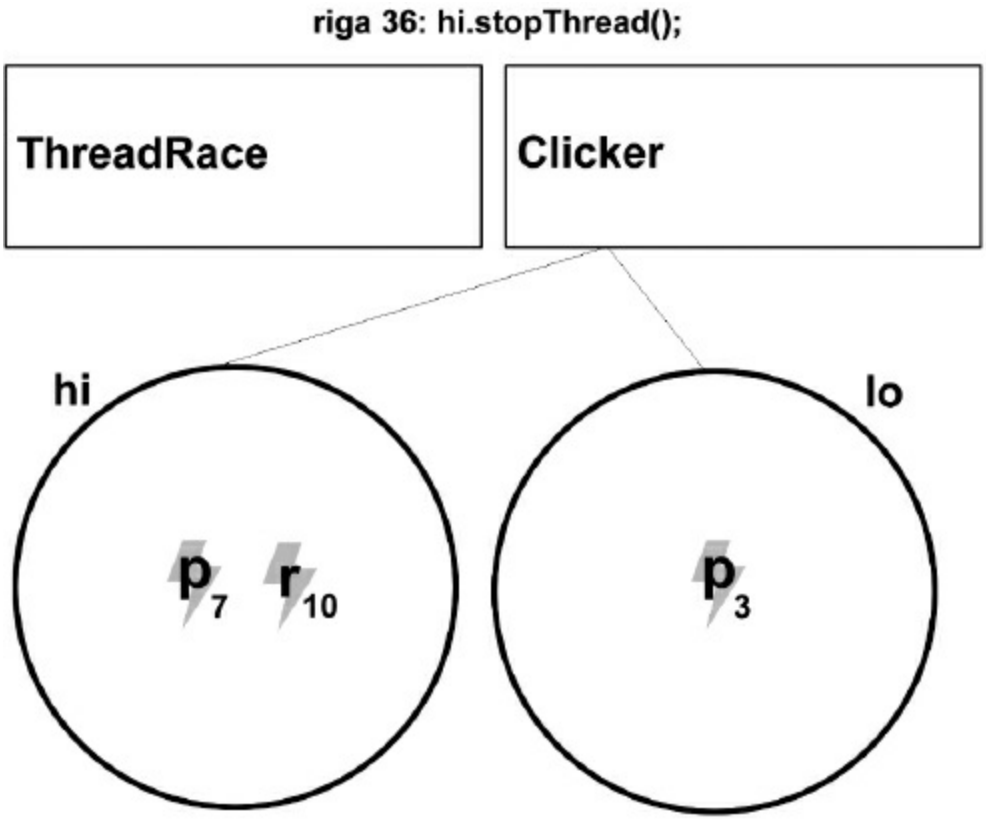


Figura 15.11 - Stop del thread hi.

Il thread principale termina la sua esecuzione stampando il risultato finale (figure 15.12 e 15.13).

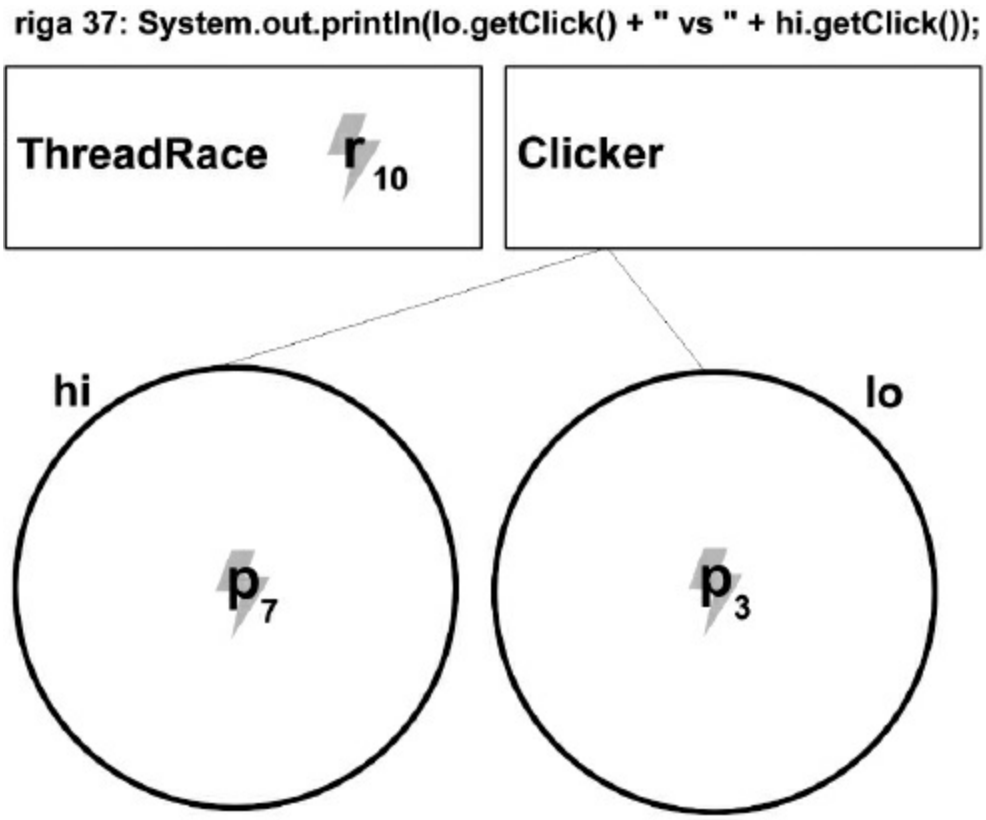


Figura 15.12 - Stampa risultato finale.

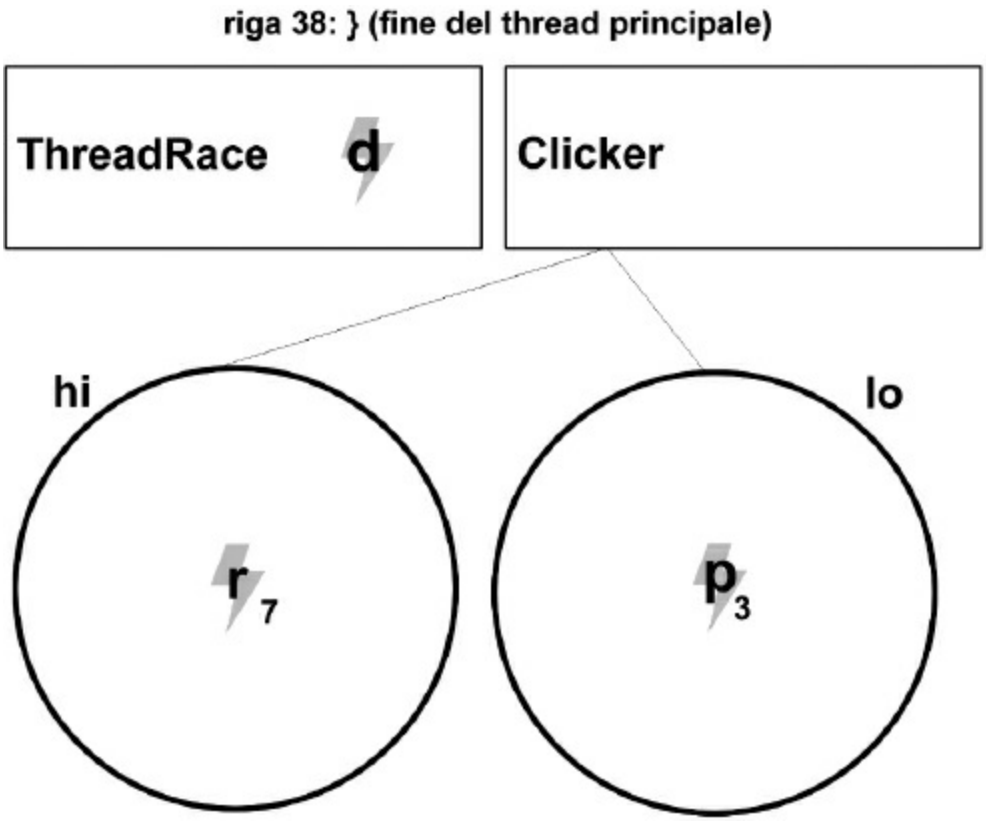


Figura 15.13 - Fine del thread principale.

A questo punto parte il thread a priorità 7, la cui esecuzione si era bloccata alla riga 13 o 14 o 15. Se riparte dalla riga 14, la variabile `click` viene incrementata un'ultima volta, senza però influenzare l'output dell'applicazione. Quando si troverà alla riga 13, la condizione del ciclo `while` non verrà verificata, quindi il thread a priorità 7 terminerà la sua esecuzione (figura 15.14). Stessa sorte toccherà al thread a priorità 3 (figura 15.15).

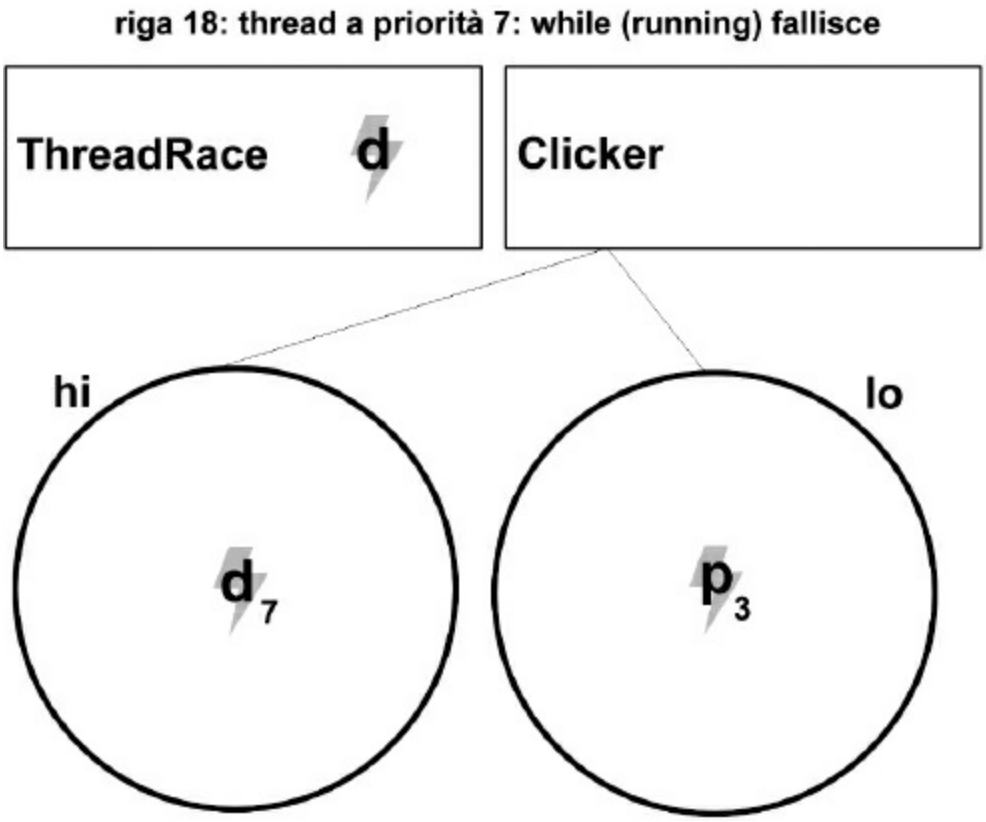


Figura 15.14 - Termina il thread hi.

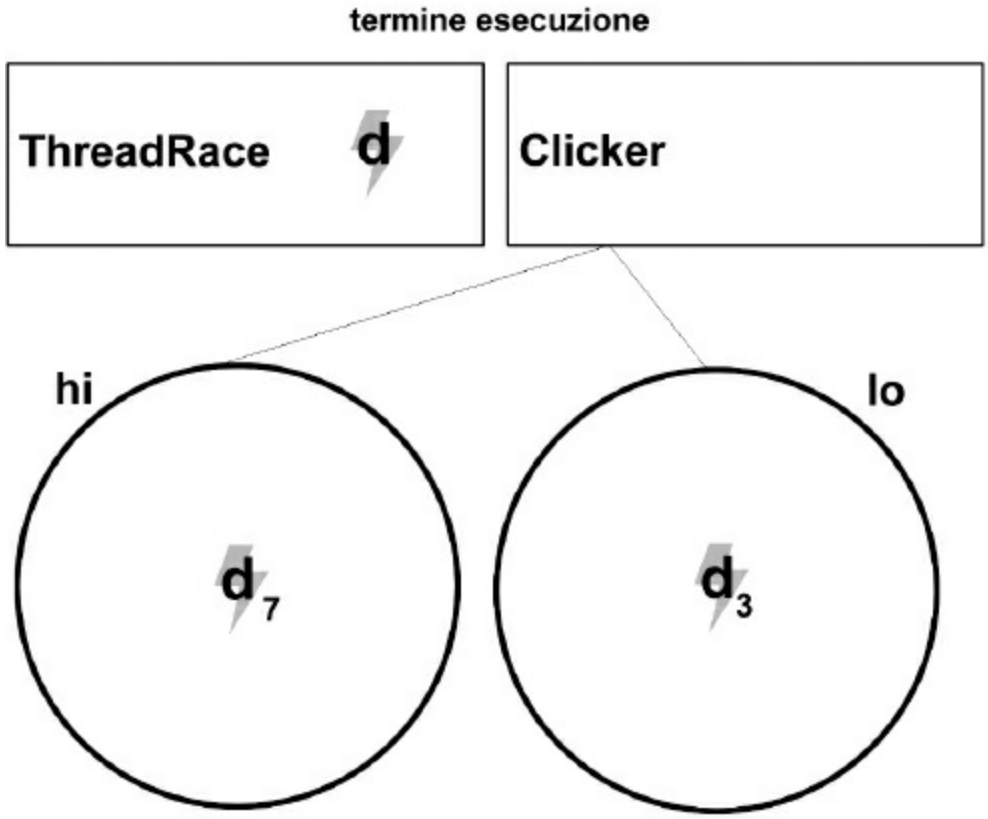


Figura 15.15 - Termina il thread lo.

15.3.4 Il modificatore `volatile`

Solo una percentuale ristretta di programmatori Java utilizza `Volatile`. Questo modificatore è applicabile solo a variabili d'istanza e per poterne apprezzare l'utilizzo, dobbiamo prima fare un piccolo preambolo. Java nel suo utilizzo della memoria esegue diverse ottimizzazioni. Tra queste c'è quella di creare copie di variabili d'istanza a cui accedono più thread, allo scopo di migliorare le prestazioni. Il modificatore `volatile` non permette questo tipo di ottimizzazione, garantendo a tutti i thread l'accesso alla stessa area di memoria. L'utilizzo di questo modificatore quindi, se da un lato può influire negativamente sulle prestazioni, in casi come quello dell'esempio precedente diventa fondamentale. Infatti dichiarando la variabile booleana `running` come `volatile`, il thread principale accede allo stesso valore della variabile `running` sia nell'oggetto `hi`, sia nell'oggetto `lo`, condividendo tali valori con i thread rispettivamente creati nel costruttore degli oggetti `hi` e `lo`. Se la

variabile booleana non fosse stata dichiarata `volatile`, è molto probabile che il programma non avrebbe funzionato correttamente, perché i thread creati nei costruttori di `hi` e `lo` non avrebbero letto la stessa variabile, modificata dal thread principale nel metodo `stopThread()`. Il programma quindi continuerebbe a vivere con i suoi thread continuando a fare clic all'infinito.

> **Si noti che da Java 5 in poi, l'accesso ad una variabile `volatile` garantirà che siano aggiornate anche tutte le altre varabili d'istanza non `volatile`, modificate dallo stesso thread. In questo esempio quindi, anche dichiarando la variabile `click` come `volatile` in luogo di `running`, avremmo ottenuto lo stesso risultato.**

Si vuole sottolineare che non è necessario dichiarare `volatile` una variabile che non sarà acceduta da più thread (anche se a volte non lo si può sapere a priori). Un altro effetto dell'uso del modificatore `volatile`, è che garantisce che gli accessi in scrittura o lettura sulla variabile siano **atomici.** Ovvero che mentre un thread legge o scrive il valore di una variabile, un altro thread deve aspettare che l'operazione si concluda per poter accedere al valore della variabile. In realtà questo è già vero per tutti i tipi primitivi (tranne che per `long` e `double`), ma l'uso del modificatore estende anche a questi due tipi questa peculiarità. Attenzione che operazioni con gli operatori di pre e post-incremento (o decremento) non sono operazioni atomiche, in quanto composte da due sotto-operazioni (aggiornamento e assegnazione) come già visto nel capitolo 4. L'atomicità in questi casi va garantita con un altro modificatore, il modificatore `synchronized` che spiegheremo nel prossimo capitolo, oppure usando la libreria `java.util.concurrent.atomic`, come vedremo nel paragrafo 15.6.3.2.

15.4 Thread e sincronizzazione

Abbiamo sino ad ora identificato un thread come un cursore immaginario. Per quanto sia utile, riteniamo il momento maturo per poter dare una definizione più scientifica di thread.

Definizione: un thread è un **processore virtuale** che esegue **codice** su determinati **dati**.

Nell'esempio precedente, `ThreadRace`, i due thread creati (quelli a priorità 7 e 3) eseguono lo stesso codice (il metodo `run()` della classe `Clicker`) utilizzando dati diversi (le variabili `lo.click` e `hi.click`). Quando però due o più thread necessitano contemporaneamente dell'accesso ad una fonte di dati condivisa, bisogna che accedano ai dati uno alla volta; cioè i loro metodi vanno **sincronizzati** `(synchronized)`. Consideriamo il seguente esempio:

```
1  class CallMe {
2      /*synchronized*/ public void call(String msg) {
3          System.out.print("[" + msg);
4          try {
5              Thread.sleep(1000);
6          }
7          catch (Exception e) {}
8          System.out.println("]");
9      }
10  }
11  class Caller implements Runnable {
12      private String msg;
13     private CallMe target;
14     public Caller(CallMe t, String s) {
15         target = t;
16         msg = s;
17         new Thread(this).start();
18     }
19     public void run() {
20         //synchronized(target) {
21             target.call(msg);
22        //}
23     }
24  }
25  public class Synch {
26      public static void main(String args[]) {
27          CallMe target = new CallMe();
28          new Caller(target, "Hello");
29          new Caller(target, "Synchronized");
30          new Caller(target, "World");
31      }
32  }
```

Output senza sincronizzazione:

```
[Hello[Synchronized[World]
]
]
```

Output con sincronizzazione:

```
[Hello]
[Synchronized]
[World]
```

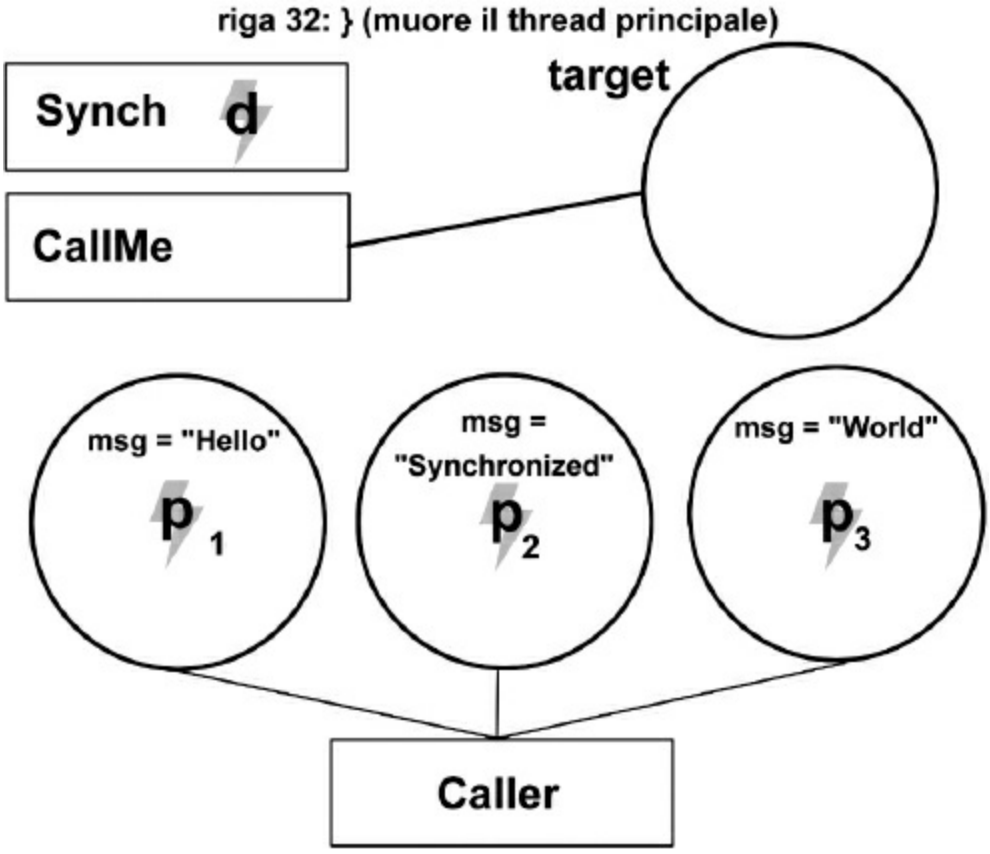


Figura 15.16 - Muore il thread principale.

15.4.1 Analisi di `Synch`

Nell'esempio ci sono tre classi: `Synch`, `Caller`, `CallMe`. Immaginiamo l'esecuzione dell'applicazione e partiamo dalla classe `Synch` che contiene il metodo `main()`. Il thread principale, alla riga 27 istanzia un oggetto chiamato `target` dalla classe `CallMe`. Alla riga 28 istanzia (senza referenziarlo) un oggetto della classe `Caller`, al cui costruttore passa l'istanza `target` e la stringa `Hello`. A questo punto il thread principale si è spostato nell'istanza

appena creata dalla classe `Caller`, per eseguirne il costruttore (righe da 14 a 18). In particolare imposta come variabili d'istanza sia l'oggetto `target` sia la stringa `Hello`, quest'ultima referenziata come `msg`. Inoltre crea e fa partire un thread al cui costruttore viene passato l'oggetto `this`. Poi il thread principale ritorna alla riga 29 istanziando un altro oggetto della classe `Caller`. Al costruttore di questo secondo oggetto vengono passate la stessa istanza `target` che era stata passata al primo, e la stringa `msg`. Prevedibilmente, il costruttore di questo oggetto appena istanziato imposterà le variabili d'istanza `target` e `msg` (con la stringa `Synchronized`), e creerà e farà partire un thread il cui campo d'azione sarà il metodo `run()` di questo secondo oggetto `Caller`. Infine, il thread principale creerà un terzo oggetto `Caller`, che avrà come variabili d'istanza sempre lo stesso oggetto `target` e la stringa `World`. Anche in questo oggetto viene creato e fatto partire un thread, il cui campo d'azione sarà il metodo `run()` di questo terzo oggetto `Caller`. Subito dopo il thread principale muore e lo scheduler dovrà scegliere quale dei tre thread in stato *ready* (pronto per l'esecuzione) dovrà essere eseguito (figura 15.16).

Avendo tutti i thread la stessa priorità 5 di default, sarà scelto quello che è da più tempo in attesa. Ecco che allora il primo thread creato (nell'oggetto dove `msg = "Hello"`) eseguirà il proprio metodo `run()`, chiamando il metodo `call()` sull'istanza `target`. Quindi il primo thread si sposta nel metodo `call()` dell'oggetto `target` (righe da 2 a 9) e, dopo aver stampato una parentesi quadra d'apertura e la stringa `Hello` (figura 15.17), si mette a dormire per un secondo (figura 15.18).

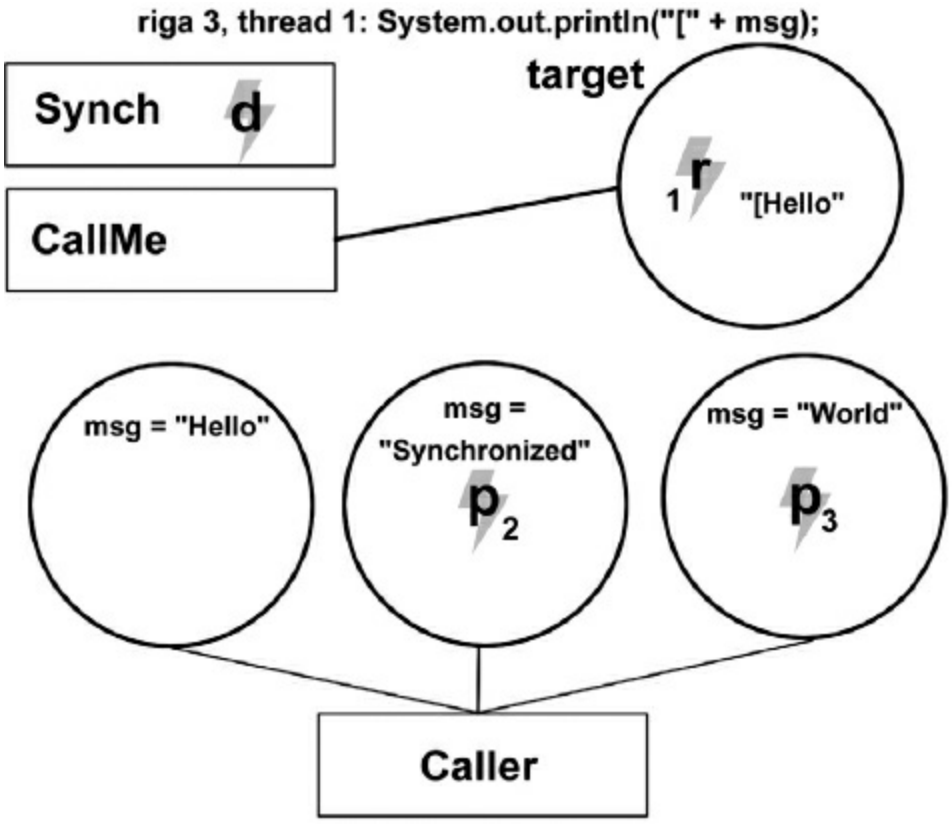


Figura 15.17 - Stampa della prima parentesi e del primo messaggio.

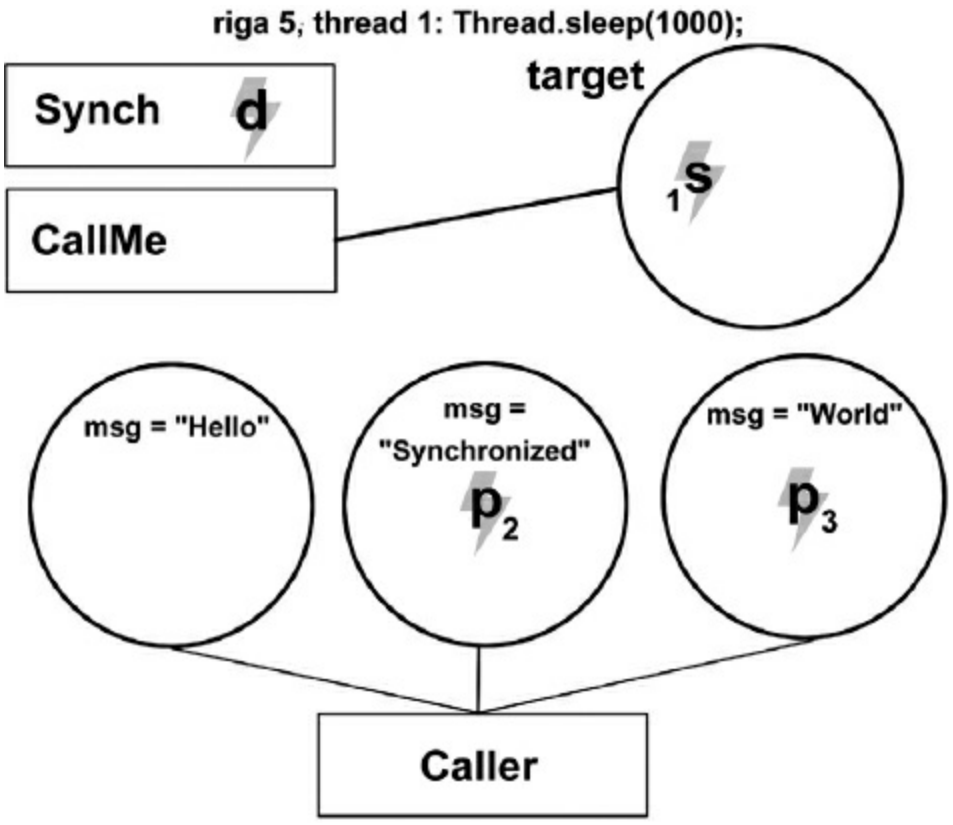


Figura 15.18 - Il thread 1 va a dormire.

A questo punto il secondo thread si impossessa del processore ripetendo in maniera speculare le azioni del primo thread. Dopo la chiamata al metodo `call()`, anche il secondo thread si sposta nel metodo `call()` dell'oggetto `target`, e, dopo aver stampato una parentesi quadra d'apertura e la stringa `Synchronized` (figura 15.19), si mette a dormire per un secondo (figura 15.20).

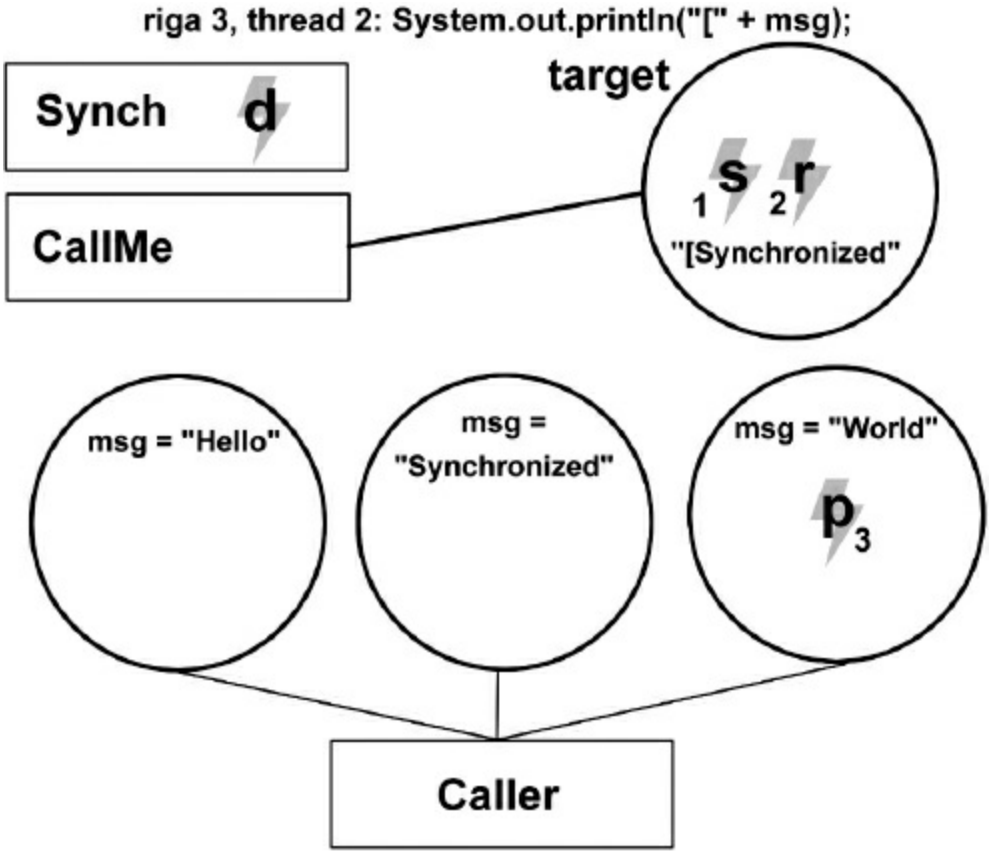


Figura 15.19 - Stampa della seconda parentesi e del secondo messaggio.

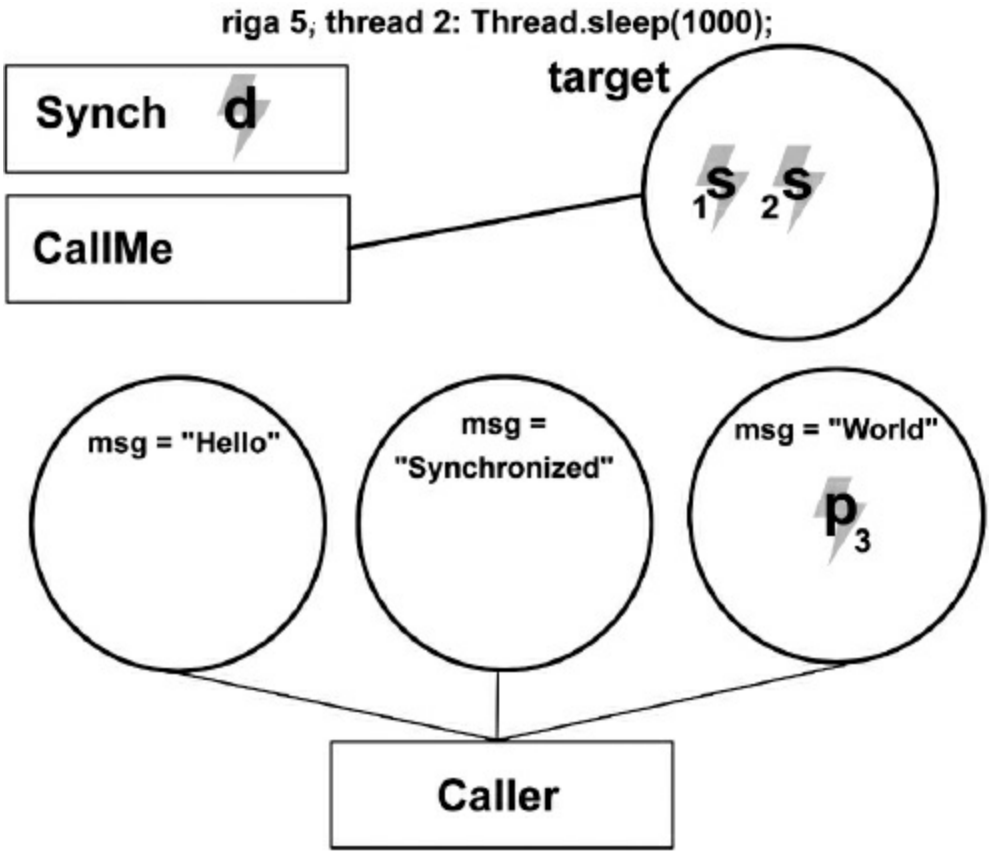


Figura 15.20 - Il thread 2 ‘va a dormire’.

Il terzo thread quindi si sposterà nel metodo `call()` dell’oggetto `target`, e anch’esso, dopo aver stampato una parentesi quadra d’apertura e la stringa `World` (figura 15.21), si mette a dormire per un secondo (figura 15.22).

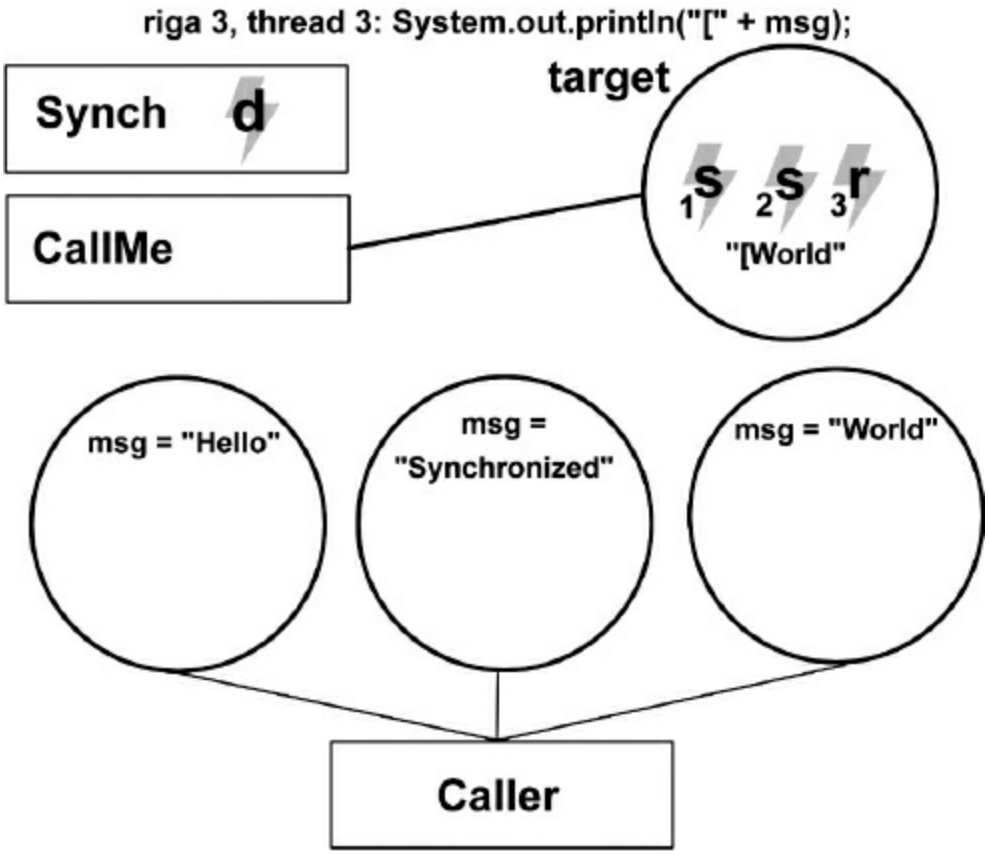


Figura 15.21 - Stampa della terza parentesi e del terzo messaggio.

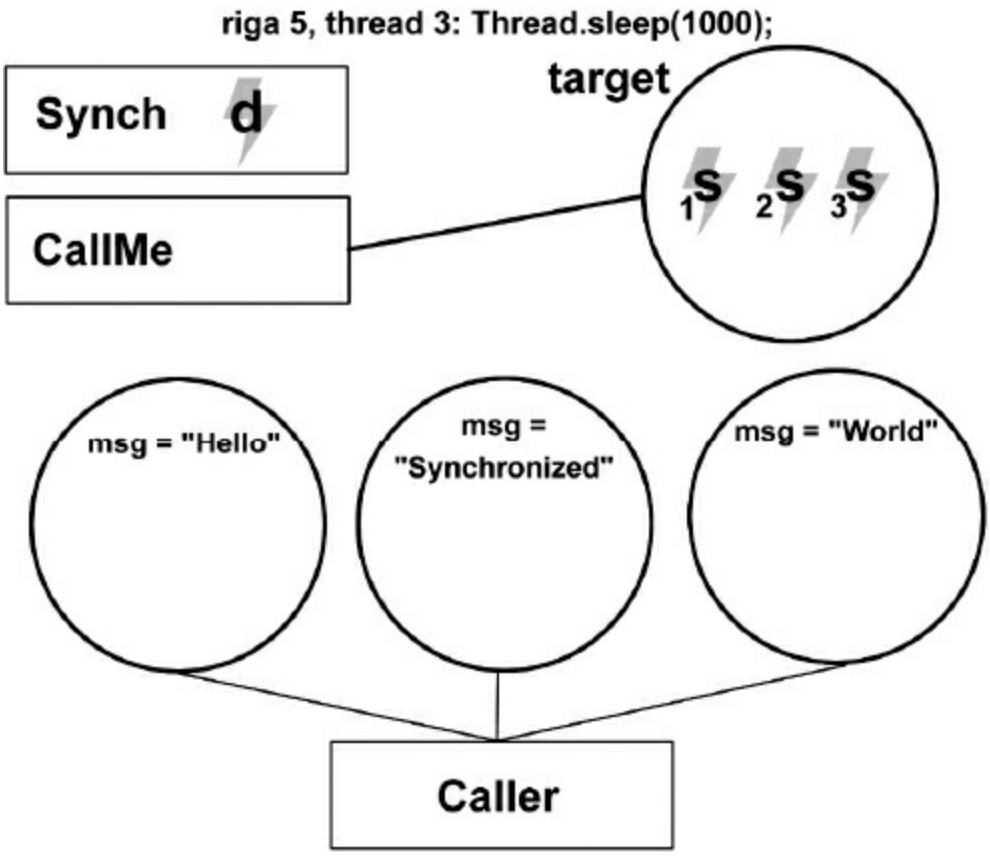


Figura 15.22 - Il thread 3 ‘va a dormire’.

Dopo poco meno di un secondo si risveglierà il primo thread, che terminerà la sua esecuzione stampando una parentesi quadra di chiusura ed andando a capo (metodo `println()` alla riga 8) (figura 15.23).

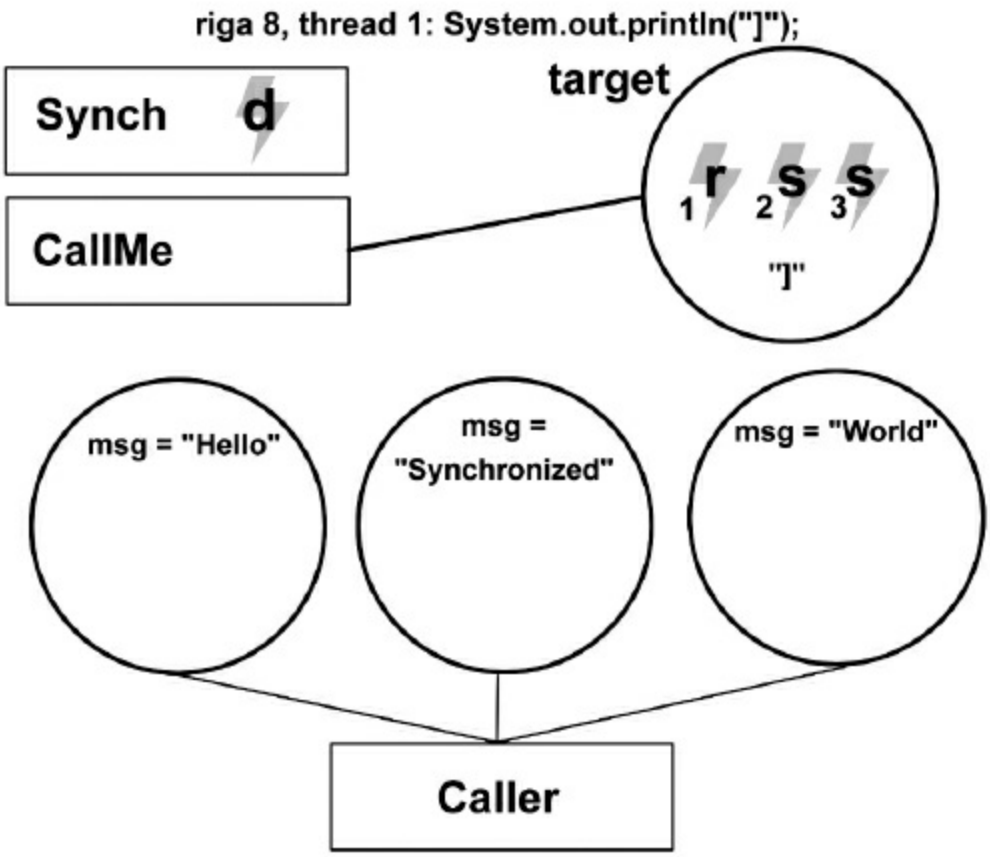


Figura 15.23 - Stampa prima parentesi di chiusura da parte del thread 1.

Anche gli altri due thread si comporteranno allo stesso modo negli attimi successivi (figure 15.24 e 15.25), producendo l'output non sincronizzato.

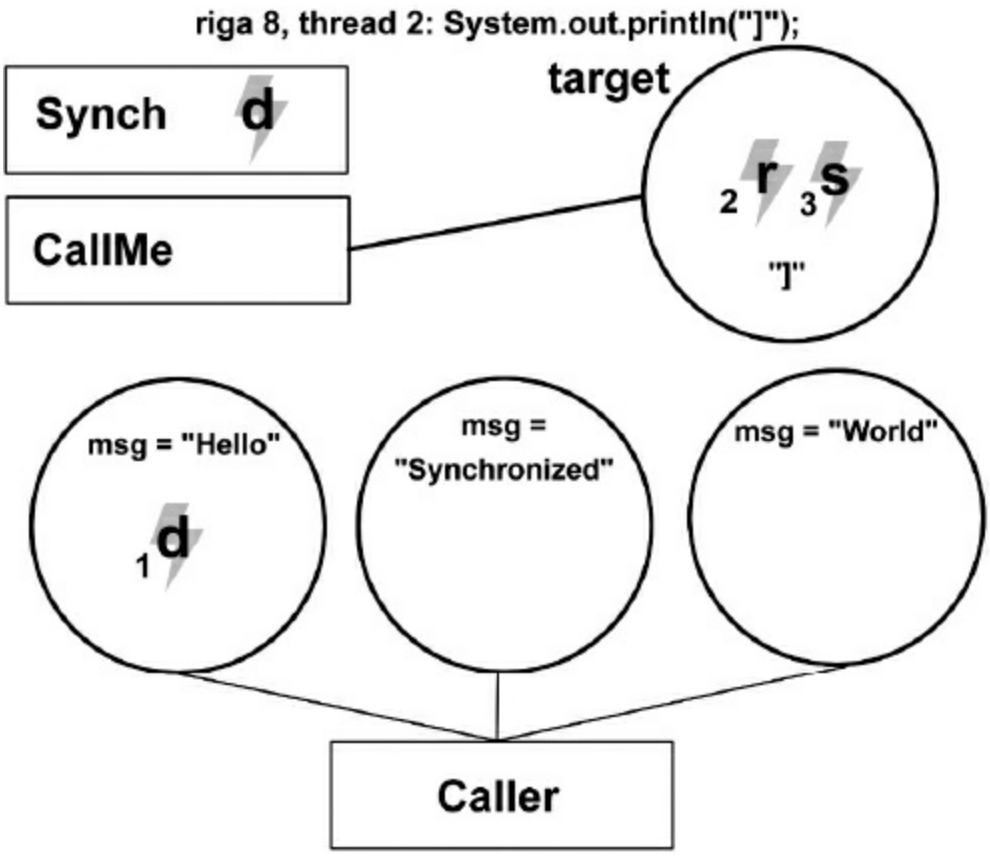


Figura 15.24 - Stampa seconda parentesi di chiusura da parte del thread 2.

L’applicazione ha una durata che si può quantificare in circa un secondo. Per ottenere l’output sincronizzato basta decommentare il modificatore `synchronized` anteposto alla dichiarazione del metodo `call()` (riga 2). Infatti, quando un thread inizia ad eseguire un metodo dichiarato sincronizzato, anche in caso di chiamata al metodo `sleep()` non lascia il codice a disposizione di altri thread. Quindi, sino a quando non termina l’esecuzione del metodo sincronizzato, il secondo thread non può eseguire il codice dello stesso metodo. Lo stesso discorso si ripete con il secondo ed il terzo thread. L’applicazione quindi ha una durata quantificabile in circa tre secondi e produce un output sincronizzato.

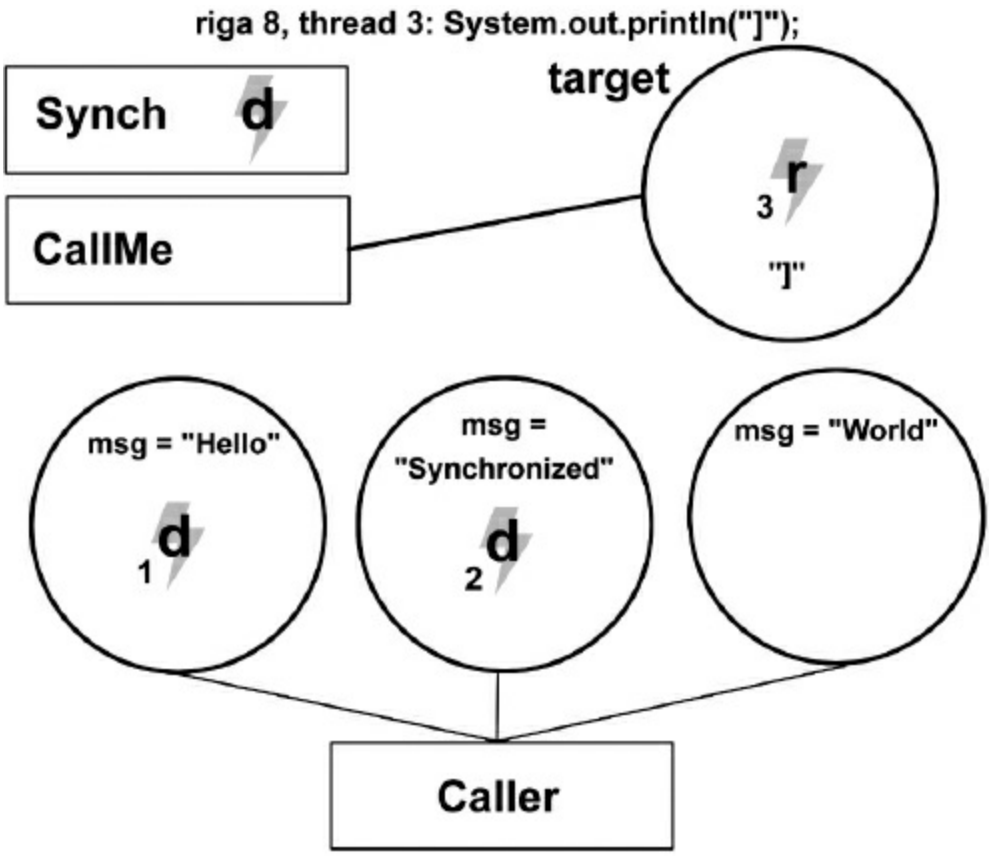


Figura 15.25 - Stampa terza parentesi di chiusura da parte del thread 3.

In alternativa, lo stesso risultato si può ottenere decommentando le righe 20 e 22. In questo caso la keyword `synchronized` assume il ruolo di comando, tramite la sintassi:

```
synchronized (nomeOggetto) {
    //...blocco di codice sincronizzato...
}
```

e quando un thread si trova all'interno del blocco di codice valgono le regole di sincronizzazione sopra menzionate. Si tratta di un modo di sincronizzare gli oggetti che da un certo punto di vista può risultare più flessibile, anche se più complesso e meno chiaro. Infatti in questo modo è possibile utilizzare un metodo di un oggetto in maniera sincronizzata o meno a seconda del contesto.

15.4.2 Monitor e lock

Esiste una terminologia ben precisa riguardante la sincronizzazione dei thread. Nei vari testi che riguardano l'argomento, viene definito il concetto di **monitor di un oggetto.** In Java ogni oggetto ha associato il proprio monitor, se contiene codice sincronizzato. A livello concettuale un monitor è un oggetto utilizzato come blocco di mutua esclusione per i thread, il che significa che solo un thread può *entrare* in un monitor in un determinato istante.

Java non implementa fisicamente il concetto di monitor di un oggetto, ma questo è facilmente associabile alla parte sincronizzata dell'oggetto stesso. In pratica, se un thread `t1` entra in un metodo sincronizzato `ms1()` di un determinato oggetto `o1`, nessun altro thread potrà entrare in alcun metodo sincronizzato dell'oggetto `o1`, sino a quando `t1` non avrà terminato l'esecuzione del metodo `ms1` (ovvero non avrà abbandonato il monitor dell'oggetto). In particolare si dice che il thread `t1` ha il **lock dell'oggetto** `o1` quando è entrato nel suo monitor (parte sincronizzata).

15.5 La comunicazione fra thread

Nell'esempio precedente abbiamo visto come la sincronizzazione di thread che condividono gli stessi dati sia facilmente implementabile. Purtroppo le situazioni che si presenteranno dove bisognerà gestire la sincronizzazione di thread non saranno sempre così semplici. Come vedremo nel prossimo esempio, la sola keyword `synchronized` non sempre basta a risolvere i problemi di sincronizzazione fra thread. Sarà necessario riuscire a far **comunicare i thread.** Descriviamo lo scenario del prossimo esempio. Vogliamo creare una semplice applicazione che simuli la situazione economica ideale, dove c'è un produttore che produce un prodotto e un consumatore che lo consuma. In questo modo, il produttore non avrà bisogno di un magazzino. Le attività del produttore e del consumatore saranno processate da due thread eseguiti in attività parallele.

```
   // Classe Magazzino ****************************
1  public class WareHouse {
2      private int numberOfProducts;
3      private int idProduct;
4      public synchronized void put(int idProduct) {
5          this.idProduct = idProduct;
```

```
6           numberOfProducts++;
7           printSituation("Produced " + idProduct);
8       }
9       public synchronized int get() {
10          numberOfProducts--;
11          printSituation("Consumed " + idProduct);
12          return idProduct;
13      }
14      private synchronized void printSituation(String msg) {
15          System.out.println(msg +"\n" + numberOfProducts
16            + " Product in Warehouse");
17      }
18  }
19
20  //classe Produttore ***************************
21
22  public class Producer implements Runnable {
23      private WareHouse wareHouse;
24      public Producer(WareHouse wareHouse) {
25          this.wareHouse = wareHouse;
26          new Thread(this, "Producer").start();
27      }
28      public void run() {
29          for (int i = 1; i <= 10; i++) {
30              wareHouse.put(i);
31          }
32      }
33  }
34
35  //classe Consumatore **************************
36  public class Consumer implements Runnable {
37      private WareHouse wareHouse;
38      public Consumer(WareHouse wareHouse) {
39          this.wareHouse = wareHouse;
40          new Thread(this, "Consumer").start();
41      }
42      public void run() {
43          for (int i = 0; i < 10;) {
44              i = wareHouse.get();
45          }
46      }
48  }
49
50  //classe del main *************************
51  public class IdealEconomy {
```

```
 52      public static void main(String args[]) {
 53          WareHouse wareHouse = new WareHouse();
 54           new Producer(wareHouse);
 55           new Consumer(wareHouse);
 56      }
 57  }
```

Ouput su Sun Solaris 8:

```
solaris% java IdealEconomy
Produced 1
1 Product in Warehouse
Produced 2
2 Product in Warehouse
Produced 3
3 Product in Warehouse
Produced 4
4 Product in Warehouse
Produced 5
5 Product in Warehouse
Produced 6
6 Product in Warehouse
Produced 7
7 Product in Warehouse
Produced 8
8 Product in Warehouse
Produced 9
9 Product in Warehouse
Produced 10
10 Product in Warehouse
Consumed 10
9 Product in Warehouse
```

15.5.1 Analisi di `IdealEconomy`

Visto l'output prodotto dal codice l'identificatore della classe `IdealEconomy` suona un tantino ironico!

> **L'output in questione è stato generato su un sistema Unix (comportamento preemptive) ed è l'output obbligato per ogni esecuzione dell'applicazione. C'è da dire che se l'applicazione fosse stata eseguita su Windows l'output**

sarebbe potuto variare drasticamente da esecuzione a esecuzione. Su Windows l'output migliore sarebbe proprio quello che è standard su Unix. Il lettore può eseguire l'applicazione più volte su un sistema Windows per conferma.

Come al solito cerchiamo di immaginare l'esecuzione dell'applicazione, supponendo di eseguire l'applicazione su un sistema Unix (caso più semplice). La classe `IdealEconomy` fornisce il metodo `main()`, quindi partiremo dalla riga 53, dove viene istanziato un oggetto `WareHouse` (letteralmente "magazzino").

La scelta dell'identificatore `magazzino` **può non essere condivisa da qualcuno. Abbiamo deciso di pensare ad un magazzino perché il nostro obiettivo è tenerlo sempre vuoto.**

Nelle successive due righe vengono istanziati un oggetto `Producer` ed un oggetto `Consumer`, ai cui costruttori viene passato lo stesso oggetto `WareHouse` (già si intuisce che il magazzino sarà condiviso fra i due thread). Sia il costruttore di `Producer` sia il costruttore di `Consumer`, dopo aver impostato come variabile d'istanza l'istanza comune di `WareHouse`, creano un thread (con nome rispettivamente `Producer` e `Consumer`) e lo fanno partire. Una volta che il thread principale ha eseguito entrambi i costruttori muore e, poiché è stato fatto partire per primo, il thread `Producer` passa in stato di esecuzione all'interno del suo metodo `run()`. Da questo metodo viene chiamato il metodo sincronizzato `put()` sull'oggetto `wareHouse`. Essendo questo metodo sincronizzato, ne è garantita l'atomicità dell'esecuzione, ma nel contesto corrente ciò non basta a garantire un corretto comportamento dell'applicazione. Infatti, il thread `Producer` chiamerà il metodo `put()` dieci volte (riga 29), per poi terminare il suo ciclo di vita e lasciare l'esecuzione al thread `Consumer`. Questo eseguirà un'unica volta il metodo `get()` dell'oggetto `wareHouse`, per poi terminare analogamente il proprio ciclo di vita e con esso il ciclo di vita dell'applicazione.

Sino ad ora non abbiamo ancora visto meccanismi chiari per far comunicare i thread. Il metodo `sleep()`, le priorità e la parola chiave `synchronized` non sono sufficienti. Contrariamente a quanto ci si possa aspettare, questi meccanismi non sono definiti nella classe `Thread`, bensì nella classe `Object`. Trattasi di metodi dichiarati `final` nella classe `Object`, pertanto ereditati da tutte le classi e non modificabili (applicando l'override). Possono essere invocati in un qualsiasi oggetto all'interno di codice sincronizzato. Questi metodi sono:

- `wait()`: comunica al thread corrente (quello che legge la chiamata a questo metodo) di abbandonare il monitor e porsi in pausa finché qualche altro thread non entra nello stesso monitor e chiama `notify()`;
- `notify()`: richiama dallo stato di pausa il primo thread che ha chiamato `wait()` nello stesso oggetto;
- `notifyAll()`: richiama dalla pausa tutti i thread che hanno chiamato `wait()` in quello stesso oggetto. Viene fra questi eseguito per primo quello a più alta priorità. Quest'ultima affermazione potrebbe non essere verificata da differenti versioni della JVM.

Esistono metodi nella classe `Thread` che realizzano una comunicazione tra thread: `suspend()` e `resume()`. Questi però sono attualmente deprecati per colpa dell'alta probabilità di produrre deadlock. Il deadlock è una condizione di errore difficile da risolvere, in cui due thread si trovano in una situazione di reciproca dipendenza in due oggetti sincronizzati. Un semplice esempio: il thread `t1` si trova nel metodo sincronizzato `m1` dell'oggetto `o1` (di cui quindi possiede il lock) e il thread `t2` si trova nel metodo sincronizzato...`m2` dell'oggetto `o2` (di cui quindi possiede il lock). Se il thread `t1` prova a chiamare il metodo `m2`, si bloccherà in attesa che il thread `t2` rilasci il lock dell'oggetto `o2`. Se anche il thread `t2` prova a chiamare il metodo `m1`, si bloccherà in attesa che il thread `t1` rilasci il lock dell'oggetto

o1. **L’applicazione rimarrà bloccata in attesa di una interruzione da parte dell’utente. Come si può intuire anche dai soli identificatori, i metodi** `suspend()` **e** `resume()`**, più che realizzare una comunicazione tra thread di pari dignità, implicavano l’esistenza di thread “superiori” che avevano il compito di gestirne altri. Poiché si tratta di metodi deprecati, non aggiungeremo altro.**

Allo scopo di far comportare correttamente l’applicazione `IdealEconomy`, modifichiamo solamente il codice della classe `WareHouse`. In questo modo sarà la stessa istanza di questa classe (ovvero il contesto di esecuzione condiviso dai due thread) a stabilire il comportamento corretto dell’applicazione. Quindi l’oggetto `wareHouse` bloccherà (`wait()`) il thread `Producer` una volta realizzato un `put()` del prodotto, dando via libera al `get()` del `Consumer`. Poi notificherà (`notify()`) l’avvenuta consumazione del prodotto al `Producer` e bloccherà (`wait()`) il `get()` di un prodotto (che non esiste ancora) da parte del thread `Consumer`. Dopo che il `Producer` avrà realizzato un secondo `put()` del prodotto, sarà prima avvertito (`notify()`) il `Consumer` e poi bloccato il `Producer` (`wait()`). Il ciclo si ripete per dieci iterazioni. Per realizzare l’obiettivo si introduce la classica tecnica del flag (boolean `empty` che vale `true` se il magazzino è vuoto). Di seguito troviamo il codice della nuova classe `WareHouse`:

```
1  public class WareHouse{
2      private int numberOfProducts;
3      private int idProduct;
4      private boolean empty = true; // magazzino vuoto
5      public synchronized void put(int idProduct) {
6          if (!empty) // se il magazzino non è vuoto...
7              try {
8                  wait(); // fermati Producer
9              }
10             catch (InterruptedException exc) {
11                 exc.printStackTrace();
12            }
13         this.idProduct = idProduct;
14         numberOfProducts++;
15         printSituation("Produced " + idProduct);
```

```
16          empty = false;
17          notify(); // svegliati Consumer
18       }
19       public synchronized int get() {
20            if (empty) // se il magazzino è vuoto...
21               try {
22                   wait(); // bloccati Consumer
23               }
24               catch (InterruptedException exc) {
25                   exc.printStackTrace();
26               }
27            numberOfProducts--;
28            printSituation("Consumed " + idProduct);
29            empty = true; // il magazzino ora è vuoto
30            notify(); // svegliati Producer
31            return idProduct;
32       }
33       private synchronized void printSituation(String msg)
34           System.out.println(msg +"\n" + numberOfProducts +
35            " Product in Warehouse");
36     }
37  }
```

Immaginiamo il runtime relativo a questo codice. Il thread `Producer` chiamerà il metodo `put()` passandogli `1` alla prima iterazione. Alla riga 6 viene controllato il flag `empty`; siccome il suo valore è `false`, non sarà chiamato il metodo `wait()`. Quindi viene impostato l'`idProduct`, incrementato il `numberOfProducts`, e chiamato il metodo `printSituation()`. Il flag `empty` viene impostato a `true` (dato che il magazzino non è più vuoto) e viene invocato il metodo `notify()`. Quest'ultimo non ha alcun effetto dal momento che non ci sono altri thread in attesa nel monitor di questo oggetto. Viene richiamato il metodo `put()` dal thread `Producer` con argomento `2`. Questa volta il controllo alla riga 6 viene verificato (`empty` è `false`, cioè il magazzino non è vuoto) e quindi il thread `Producer` chiama il metodo `wait()` rilasciando il lock dell'oggetto. A questo punto il thread `Consumer` esegue il metodo `get()`. Il controllo alla riga 20 fallisce perché `empty` è `false`, quindi non viene chiamato il metodo `wait()`. Viene decrementato `numberOfProducts` a `0` e chiamato il metodo `printSituation()`; si imposta il flag `empty` a `true`. Poi viene chiamato il metodo `notify()`, che toglie il thread `Producer` dallo stato di `wait()`. Viene restituito il valore di

`idProduct`, che è ancora `1`. Poi viene richiamato il metodo `get()`, ma il controllo alla riga 20 è verificato e dunque il thread `Consumer` si mette in stato di `wait()`. Quindi il thread `Producer` continua la sua esecuzione da dove si era bloccato cioè dalla riga 8. Viene impostato `idProduct` a `2`, incrementato di nuovo ad `1` `numberOfProducts` e chiamato il metodo `printSituation()`. Il flag `empty` viene impostato a `false` e viene chiamato il metodo `notify()`, che risveglia il thread `Consumer`.

15.6 Concorrenza

Quando abbiamo a che fare con concetti tanto complicati, è sconsigliato re-inventarsi ogni volta l'acqua calda. Benché a volte ci sia la convinzione di avere la situazione sotto controllo, con i thread non si è mai troppo sicuri. Inoltre i bachi creati dall'utilizzo di thread sono molto difficili da risolvere e spesso anche molto dannosi!

Come abbiamo già detto, la gestione dei thread introduce una nuova dimensione alla programmazione: la dimensione temporale. La *programmazione multi-threaded* è da considerarsi un nuovo tipo di programmazione. Non è la tipica programmazione fatta solo di `if` e `for`, ed esistono interi libri che parlano solo di questo argomento. Durante la lettura di queste pagine abbiamo già parlato di altri tipi di programmazione: prima della *programmazione object oriented*, poi della *programmazione per contratto*, poi della *programmazione generica*, e quando parleremo di espressioni lambda nel prossimo capitolo, approcceremo ad un altro tipo di programmazione ancora. Ogni volta è un po' come partire da zero e la cosa migliore da fare è sfruttare le tecniche e i design pattern conosciuti. Per esempio nel codice di `IdealEconomy` abbiamo usato una tecnica nota come **guarded block**, per gestire la comunicazione nella classe `Warehouse` tra `Producer` e `Consumer`. È grazie ad essa che abbiamo pensato a chiamare il metodo `wait()` in un ciclo, e a far dipendere la chiamata a `notify()` da un booleano, non ci siamo inventati niente. Proprio per questo motivo in Java esistono librerie che mettono a disposizione implementazioni pronte per l'uso, che risolvono la maggior parte dei tipici problemi che si presentano nella programmazione multi-threaded. Tuttavia senza conoscere l'argomento è difficile utilizzare questi strumenti in maniera consapevole.

15.6.1 Oggetti Immutabili

Durante lo studio di questo libro, abbiamo incontrato diverse classi che abbiamo definito **immutabili.** Con questo aggettivo abbiamo inteso che gli oggetti istanziati da queste classi non potevano cambiare il loro stato interno. È il caso delle classi `Wrapper, Math, String` e di tutte le classi della libreria Date-Time API. Il vantaggio di usare oggetti immutabili in luogo degli oggetti mutabili, è che gli oggetti immutabili sono **thread-safe** (in italiano **sicuri per l'utilizzo dei thread**). È un concetto semplice da comprendere dopo quello che abbiamo visto nei precedenti paragrafi: un oggetto thread-safe non corre il rischio di essere modificato internamente da più thread concorrenti contemporaneamente. Per esempio due thread `t1` e `t2` potrebbero accedere contemporaneamente all'oggetto `o1` che contiene una variabile `value` che al momento della lettura dei due thread vale `0`. Poi entrambi provano a modificare la variabile con lo scopo di incrementarla di un'unità. Supponiamo che il thread `t1` arrivi prima e modifichi il valore di `value` a `1`. Quando arriverà il thread `t2` incrementerà il valore di `value` a `2`, ma l'obiettivo di `t2` era quello di incrementare il valore a `1` dal momento che al momento della lettura `value` valeva `0`. Questo succede perché `o1` non è un oggetto thread-safe.

Un oggetto immutabile invece è thread-safe visto che il suo stato interno non è modificabile. Inoltre gli oggetti immutabili garantiscono altri vantaggi relativi al loro trattamento in memoria da parte della virtual machine. Le **linee guida** per creare una classe immutabile sono le seguenti:

1. tutti gli attributi devono essere dichiarati `final` e `private`;
2. non fornire alla classe metodi di tipo setter, o che modifichino le variabili d'istanza in generale;
3. nel caso i valori delle variabili d'istanza di tipo reference debbano essere passati in input a metodi dall'esterno, bisogna preoccuparsi di creare delle copie da immagazzinare internamente, per evitare che siano modificate dall'esterno successivamente;
4. nel caso variabili interne di tipo reference debbano essere esposte tramite metodi getter, bisogna preoccuparsi di creare delle copie da restituire all'esterno, per evitare che siano modificate dall'esterno

successivamente;

5. impedire alle sottoclassi di fare override dei metodi. Quindi dichiarare `final` la classe dovrebbe essere sufficiente. Dichiarare `final` anche i metodi metterebbe ancora più in risalto le nostre intenzioni.

Se vogliamo modificare un oggetto immutabile dobbiamo creare dei metodi che restituiscono altri oggetti modificabili a partire dall'originale. Un buon esempio sono i metodi di `String` come `toUpperCase()`:

```
String nome = "Simone";
String nomeMaiuscolo = nome.toUpperCase();
System.out.println(nome);
System.out.println(nomeMaiuscolo);
```

Questo codice stamperà:

```
Simone
SIMONE
```

in quanto la variabile `nome` non è cambiata con la chiamata del metodo `toUpperCase()`. Di seguito un esempio di classe immutabile e quindi thread-safe:

```
import java.util.*;

public class ImmutableObject {
    private final int integer; //variabile d'istanza
primitiva
    private final String string; //variabile d'istanza
complessa immutabile
    private final Calendar calendar; //variabile d'istanza
complessa
    //Costruttore che prende le variabili dall'esterno
    public ImmutableObject(int integer, String string,
Calendar calendar) {
        this.integer = integer;
        this.string = string;
        //segue punto 3
        this.calendar = (Calendar)calendar.clone();
    }
    //Questo metodo segue il punto 4
    public Calendar getDate() {
```

```
            return (Calendar)calendar.clone();
        }
        //Metodo di test
        public void stampaCalendar() {
            System.out.println(calendar);
        }
    }
```

15.6.2 Tecnologie Java e classi standard ausiliarie

La gestione dei thread in Java può considerarsi semplice se paragonata alla gestione dei thread in altri linguaggi. Inoltre una classe (`Thread`), un'interfaccia (`Runnable`), una parola chiave (`synchronized`) e tre metodi (`wait()`, `notify()` e `notifyAll()`) rappresentano il nucleo di conoscenza fondamentale per gestire applicazioni multithreaded. Ma le situazioni multithreaded che bisognerà gestire nelle applicazioni reali non saranno tutte così semplici come nell'ultimo esempio. Tuttavia la tecnologia Java fa ampio uso del multithreading. La notizia positiva è che la complessità del multithreading viene spesso gestita dalla tecnologia stessa. Consideriamo per esempio la tecnologia web Java Servlet (http://www.oracle.com/technetwork/java/javaee/servlet/index.html).

Java Servlet è in grado di gestire il multithreading in maniera automatica, allocando thread diversi per servire ogni richiesta del client. Quindi lo sviluppatore è completamente esonerato dal dover scrivere codice per la gestione dei thread. Gestiscono thread automaticamente anche altre tecnologie Java Web (per esempio Java Server Pages), Enterprise (per esempio Enterprise JavaBeans...) e Standard (per esempio le interfacce grafiche). Poi ci sono alcune classi della libreria Java (`java.nio.channels.SocketChannel`, `java.util.Timer`, `java.util.concurrent.ThreadLocalRandom` etc.) che semplificano enormemente il rapporto tra lo sviluppatore e i thread. Per esempio prendiamo la classe `TimerTask`. Essa ci permette di creare un semplice thread che si deve risvegliare dopo un certo lasso di tempo, per eseguire un determinato compito. Per esempio, questa classe viene utilizzata da EJE per creare la funzionalità di sveglia. In particolare bisogna fare due semplici cose:

1. creare una sottoclasse della classe astratta `TimerTask` (la quale a sua volta implementa l'interfaccia `Runnable`), che definirà un metodo

`run()` chiamato al risveglio del `Timer`;

2. schedulare l'evento passando un oggetto `TimerTask` al metodo `schedule()` di un oggetto `Timer`.

Riguardo il passo 2, ecco come EJE schedula un evento sveglia (codice modificato per semplicità):

```
new java.util.Timer().schedule(
  new AlarmClockTimerTask(messaggio), millisecondi);
```

dove `AlarmClockTimerTask` è l'implementazione del `TimerTask, messaggio` è una stringa da stampare e `millisecondi` è il numero di millisecondi che devono passare affinché il `TimerTask` sia eseguito. Semplicissimo e pulito. Naturalmente è possibile schedulare anche più di un `TimerTask`, ed esistono diversi overload del metodo `schedule()` della classe `Timer`, per schedulare i `TimerTask` ad intervalli regolari usando un oggetto `Date`, e così via. Esistono anche una serie di classi ed interfacce del framework Collections, che servono proprio per essere utilizzate in ambienti multithreading. Le varie `Queue, BlockingQueue` e `ConcurrentMap` saranno affrontate nel capitolo 17 dedicato alle collezioni.

15.6.3 Libreria Java sulla concorrenza

Dalla versione 5 di Java il supporto al multithreading è stato ulteriormente ampliato, con l'introduzione di classi molto potenti che semplificano il lavoro dello sviluppatore. Con il package `java.util.concurrent` ed i suoi sotto-package, Java definisce una serie di strumenti di alto livello per gestire la concorrenza tra thread in maniera semplice ed elegante.

15.6.3.1 II package `java.util.concurrent.locks`

II package `java.util.concurrent.locks` definisce una serie di classi ed interfacce che consentono di gestire in maniera più semplice i lock, rispetto ai vari `synchronized, wait()` e `notify()` che abbiamo visto negli esempi precedenti. L'interfaccia principale si chiama `Lock`, e definisce tra i vari metodi due versioni (overload) del metodo `tryLock()`. Una di queste prende in input dei valori per specificare un time-out affinché un thread esca

dall'impasse nel caso non si riesca a prendere il lock di un oggetto. La più famosa delle implementazioni di `Lock` si chiama `ReentrantLock`, e nel seguente esempio si può facilmente comprendere come funziona:

```
import java.util.concurrent.locks.*;
import java.util.concurrent.*;

public class LockExample implements Runnable {
    private int numeroIntero;
    private Lock lock;

    public LockExample(int numeroIntero) {
        this.numeroIntero = numeroIntero;
        this.lock = new ReentrantLock();
}

@Override
public void run() {
    try {
        if (lock.tryLock(5, TimeUnit.SECONDS)) {
            numeroIntero++;
            System.out.println("In lock " + numeroIntero);
       }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
         lock.unlock();
    }
    System.out.println("Out lock " + numeroIntero);
    }
}
```

All'interno di questa classe che implementa `Runnable` (e quindi codice eseguibile per thread) nel metodo `run()` un oggetto `lock` prova a prendere il lock dell'oggetto corrente, con un time-out di cinque secondi. Se vi riesce entra nel blocco di codice dell'`if`, prende il lock dell'oggetto e lo rilascia nella clausola `finally`. A differenza della parola chiave `synchronized`, un oggetto `Lock` ha quindi la possibilità di impostare un tempo di time-out, e di specificare un `unlock()` nel codice di `finally`. Un altro punto a favore dell'utilizzo di un `ReentrantLock` rispetto ad un blocco sincronizzato, è quello di poter creare lock con **fairness**, che in italiano potremo tradurre con

“correttezza”. Stiamo parlando di un semplice booleano che può essere passato in input al costruttore di `ReentrantLock`, denominato `fair`. Se impostato a `true`, questo booleano permetterà allo scheduler della JVM di creare una gerarchia di priorità di esecuzione tra thread, basata sul tempo di creazione del `ReentrantLock`. Prima è stato creato l’oggetto lock, e maggiore è la sua priorità. Quindi, nel caso in un certo istante sia rilasciato il lock di un oggetto, e diversi thread abbiano la possibilità di prendere questo lock, sarà favorito il più vecchio. Con la sincronizzazione ordinaria non è possibile creare tale gerarchia. Uno svantaggio del `ReentrantLock` invece, può essere considerato proprio l’obbligo di dover utilizzare un blocco `try-catch-finally` (con il catch facoltativo), il che rende il codice pesante e verboso. Fortunatamente gli IDE ci vengono incontro con la creazione automatica dei blocchi `try-catch-finally`.

Nello stesso package esistono anche altre interessanti tipi come l’interfaccia `Condition` e `StampedLock`. Mentre possiamo considerare `Lock` come un’alternativa ad un blocco o un metodo `synchronized`, `Condition` può essere considerata come l’alternativa all’utilizzo dei metodi per la gestione del monitor di un oggetto: `wait()`, `notify()` e `notifyAll()`. Si ottiene un’istanza di `Condition` tramite il metodo `newCondition()` chiamato su un `Lock`, e si usano i metodi `await()` per sostituire `wait()`, e `signal()` e `signalAll()` per sostituire rispettivamente `notify()` e `notifyAll()`. Esistono anche diversi overload di `await()` tra cui un paio che consentono di specificare un time-out.

La classe `StampedLock` invece, è stata introdotta solo con Java 8. Si tratta di una nuova tipologia di lock, che permette di prendere il lock di un oggetto con tre modalità differenti: *scrittura*, *lettura* e *lettura ottimistica* basata su timestamp, ovvero degli istanti calcolati come `long`, che serviranno per validare le varie azioni sull’oggetto.

15.6.3.2 II package `java.util.concurrent.atomic`

II package `java.util.concurrent.atomic` definisce una serie di classi come `AtomicLong`, `AtomicDouble` e così via, che permettono di effettuare operazioni atomiche in maniera thread-safe. Le classi di questo package definiscono metodi che combinano più operazioni come `getAndDecrement()`,

`getAndIncrement()`, `compareAndSet()` e `getAndSet()` che garantiscono che queste operazioni siano eseguite in maniera atomica, anche nel caso più thread accedessero contemporaneamente alla stessa variabile. Per esempio alla fine del paragrafo 15.3.4, abbiamo evidenziato il fatto che le operazioni di pre e post incremento (o decremento) non sono operazioni atomiche e quindi non thread-safe. Con le classi di questo package possiamo risolvere questo problema in maniera semplice ed elegante. Per esempio possiamo chiamare il metodo `incrementAndGet()` in luogo dell'operatore di preincremento:

```
AtomicLong atomicLong = new AtomicLong(10);
long incrementedLong = atomicLong.incrementAndGet();
```

Altre classi appartenenti a questo package sono gli adder (`LongAdder`, `DoubleAdder`) e gli accumulator (`LongAccumulator`, `DoubleAccumulator`). Entrambi i tipi di classi permettono di fare operazioni atomiche su più elementi. Ma si tratta di classi che apprezzeremo meglio dopo aver studiato le espressioni lambda nel prossimo capitolo.

15.6.3.3 Interfacce Executors

Nel package `java.util.concurrent` sono definite tre interfacce note come **Executors.** Si tratta di semplici interfacce che definiscono un metodo `execute()`, che consente di eseguire thread in modo molto semplice. In generale un metodo `executed` prende in input un oggetto `Runnable` ed è invocato su un oggetto executor. Ricordiamo invece che abbiamo imparato a creare thread al cui costruttore passavamo un oggetto `Runnable` e sul thread chiamavamo il metodo `start()`. La grande differenza rispetto a quanto visto sino ad ora, è che gli Executors permettono di separare nettamente il codice dei thread e degli oggetti `Runnable`. Dal punto di vista architetturale questa scelta è più che opportuna. Questo consente di gestire i nostri programmi con più semplicità, senza mischiare codice funzionale con codice per gestire la concorrenza.

Le interfacce Executors sono:

- `Executor`: una semplice interfaccia che permette l'esecuzione di thread.

- `ExecutorService`: un'interfaccia che estende `Executor` e che definisce anche un metodo `submit()`, più completo rispetto ad `execute()`. Infatti il `submit()` può prendere in input un oggetto di tipo `Callable` in luogo di un oggetto `Runnable`, e restituisce un oggetto di tipo `Future` che consente di recuperare dati in output dopo l'esecuzione del metodo. `ExecutorService` inoltre aggiunge funzionalità per gestire il ciclo di vita sia dei processi che dell'executor stesso. Vedremo nelle prossime righe un esempio che coinvolge `ExecutorService, Callable` e `Future`.
- `ScheduledExecutorService`: estende `ExecutorService` e permette di schedulare i processi.

Probabilmente l'esempio più famoso di executor, è la creazione di un **pool di thread**, da sempre una delle implementazioni più complicate da creare per un programmatore Java. Invece in Java gli oggetti `ExecutorService` consentono con poche righe di codice la creazione di pool di thread. Il pattern dei **threads pool** (abbiamo già parlato di **pool** nei paragrafi 12.5.1 riguardante le stringhe, e nel paragrafo 12.6.1.1 riguardante le classi wrapper), nella sua definizione più banale, permette di creare un pool, ovvero una *riserva*, dove un numero fisso di thread già pronti aspetta di essere usato. Quando c'è bisogno di eseguire un processo, viene richiesta la disponibilità all'executor, che restituisce la prima istanza di thread disponibile (ovvero che non sta già lavorando). Quando un thread finisce il suo compito ritorna nel pool in attesa di essere usato. Questo pattern permette di limitare il numero di istanze dei thread, e fa sì che si possa evitare la creazione di un thread a richiesta, visto che è un processo dispendioso per le risorse del programma. Inoltre un'implementazione efficiente consente performance migliori. Segue un semplice esempio di implementazione di `ExecutorService`:

```
import java.util.concurrent.*;

public class TestExecutorService {
    public static void main(String args[]) throws Exception
{
        ExecutorService service =
Executors.newFixedThreadPool(10);
        for (int i =0; i<100; i++) {
```

```
            Future<Integer> future = service.submit(new
Processo(i));
            System.out.println(" Valore : " + future.get());
        }
        service.shutdownNow();
    }
}

class Processo implements Callable<Integer> {
    private int id;

    public Processo(int id) {
        this.id = id;
    }

    @Override
    public synchronized Integer call() {
        System.out.print("ID : " + this.id + ", thread : " +
                         Thread.currentThread().getName());
        return this.id;
    }
}
```

Abbiamo creato con una classe interna un'implementazione dell'interfaccia funzionale generica `Callable`. Questa definisce il metodo SAM (cfr. paragrafi 7.7.4, 16.1.1) `call()`, che equivale al metodo `run()` di `Runnable`, ma che può restituire un tipo di ritorno definito dal tipo generico dichiarato. In questo caso il metodo stampa la situazione e restituisce un `Integer`, definito direttamente come tipo parametro di `Callable`. Possiamo invece osservare che nel metodo `main()` della classe `TestExecutorService` è istanziato un oggetto `ExecutorService` mediante il metodo `newFixedThreadPool()` che istanzia un pool non ridimensionabile di 10 elementi. Poi all'interno del ciclo `for` è invocato il metodo `submit()` (che equivale al vecchio metodo `start()` della classe `Thread`) sull'oggetto `ExecutorService`. Viene poi catturato l'oggetto `Future` con tipo parametro `Integer`, che contiene il risultato dell'operazione. È possibile creare pool di thread più complessi con `ThreadPoolExecutor` e `ScheduledThreadPoolExecutor`.

Con Java 7 è stato anche introdotto un nuovo framework detto **Fork/Join** che estende in qualche modo le funzionalità degli Executors mettendo a

disposizione dell'utente un algoritmo basato sulla ricorsione detto **work-stealing.** Con questo framework i thread del pool che hanno completato il loro compito possono "rubare", ad altri thread ancora impegnati, parti di codice da eseguire. Questo permette di ottimizzare i tempi di esecuzione in ambienti multi-processori. Il framework si basa sulla classe `ForkJoinPool` (un'estensione della classe astratta `AbstractExecutorService`) e il concetto di dividere il codice da eseguire in più parti mediante l'implementazione di una tra le classi astratte `RecursiveAction` o `RecursiveTask`. Quest'ultima ha la possibilità di conservare un valore di ritorno per l'operazione da eseguire. Una tipica implementazione del framework Fork/ Join, è il metodo `parallelSort()` della classe `Arrays`, che ordina gli elementi di un array. L'array viene suddiviso in sotto array che sono ordinati in parallelo per poi essere a loro volta ricongiunti (join) per ottenere il risultato finale. In generale questo framework viene raramente implementato dal programmatore. Piuttosto il programmatore utilizza le implementazioni fornite dalla libreria Java, come appunto il metodo `parallelSort()`.

15.6.3.4 La classe `Semaphore`

Ci sono altre classi che sono degne di nota. Per esempio in questa libreria esiste il concetto di **semaforo**, argomento già familiare a chi conosce la gestione dei thread in altri ambienti. Gli oggetti della classe `Semaphore`, concettualmente funzionano in maniera molto simile ai veri semafori. Quando è rosso (ovvero il semaforo ha acquisito il lock sull'oggetto) non si passa, quando è verde si passa (ovvero altri thread possono prendere il lock dell'oggetto). La classe `Semaphore` mette a disposizione metodi come `acquire()` e `release()` che garantiscono il lock e l'unlock del monitor di un oggetto. È anche definito il metodo `tryAcquire()` che permette di specificare un timeout massimo di attesa per prendere il lock. Come abbiamo visto anche con gli oggetti `Lock`, la gestione di un semaforo si fa preferire ai comandi `wait()` e `notify()`. In particolare i semafori consentono di restringere a un certo numero di thread, l'accesso contemporaneo ad una risorsa. Nel seguente esempio, creiamo una classe `Veicolo` che implementa `Runnable`:

```
import java.util.concurrent.Semaphore;
public class Veicolo implements Runnable {
```

```
    private Incrocio incrocio;
    public Veicolo(Incrocio incrocio) {
        this.incrocio = incrocio;
    }

    public void run() {
        incrocio.rispettaSemaforo();
    }
}
```

Questa definisce un'aggregazione con la seguente classe `Incrocio`, la quale è impostata come variabile d'istanza tramite il suo costruttore. Il suo metodo `run()` chiama il metodo `rispettaSemaforo()` sull'oggetto `incrocio`.

```
import java.util.concurrent.Semaphore;

public class Incrocio {
    Semaphore semaforo = new Semaphore(1);

    public void rispettaSemaforo() {
         try {
             semaforo.acquire();
             System.out.println(Thread.currentThread().getName(
 +
                                " sta superando
l'incrocio");
             Thread.sleep(1000);
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        } finally {
             semaforo.release();
             System.out.println(Thread.currentThread().getName(
 +
                                " ha superato l'incrocio");
        }
    }
}
```

La classe `Incrocio` definisce una variabile d'istanza di tipo `Semaphore`, al cui costruttore è stato passato il valore `1`, che rappresenta il numero di **permits** che possono essere concessi. Il numero di *permits* è il numero di permessi, implementati mediante un contatore, che mantiene l'oggetto semaforo per concedere l'accesso alla risorsa condivisa. In questo esempio un solo thread

alla volta potrà superare il metodo `acquire()`, senza che un altro thread chiami il metodo `release()` sullo stesso oggetto semaforo. Le seguenti righe di codice:

```
Incrocio incrocio = new Incrocio();
Thread t1 = new Thread(new Veicolo(incrocio),"Una Ford");
t1.start();
Thread t2 = new Thread(new Veicolo(incrocio),"Una Nissan");
t2.start();
```

consentiranno di simulare la situazione dove solo una macchina per volta supererà l'incrocio. Segue l'output:

```
Una Ford sta superando l'incrocio
Una Ford ha superato l'incrocio Una Nissan sta superando
l'incrocio
Una Nissan ha superato l'incrocio
```

15.6.3.5 La classe `CyclicBarrier`

Vale la pena dare uno sguardo anche alla classe `CyclicBarrier`. Questa fornisce la possibilità ad un gruppo di thread di sincronizzarsi e attendersi l'un l'altro in un determinato punto del codice (da qui il nome della classe). In seguito viene presentato un esempio che simula la situazione in cui, dei partecipanti ad una festa devono essere tutti presenti affinché la festa abbia inizio. Prima definiamo la classe `Festa` che implementa `Runnable` e definisce come variabile d'istanza una `CyclicBarrier`, che farà da punto di sincronizzazione all'interno del metodo `run()` tra più thread. Quando un thread arriva a chiamare il metodo `await()`, si blocca ad aspettare che altri thread arrivino allo stesso punto per poter continuare la sua esecuzione:

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class Festa implements Runnable {
    private CyclicBarrier luogoDellaFesta;

    public Festa(CyclicBarrier luogoDellaFesta) {
        this.luogoDellaFesta = luogoDellaFesta;
    }
```

```
    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName()
 +
                               " è presente");
            luogoDellaFesta.await();
            System.out.println(Thread.currentThread().getName()
 +
                               " ha iniziato a
festeggiare...");
        } catch (InterruptedException |
BrokenBarrierException ex) {
             ex.printStackTrace();
        }
    }
}
```

La classe seguente invece, crea una `CyclicBarrier` passando al suo costruttore il numero dei thread che devono chiamare `await()` per sbloccare la barriera, e un oggetto `Runnable` creato al volo sotto forma di classe anonima che definisce l'azione che deve essere eseguita dalla `CycliclBarrier` una volta sbloccata. Inoltre definisce i protagonisti della festa (i thread), e li fa partire:

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class TestFesta {
    public static void main(String args[]) {
        CyclicBarrier luogoDellaFesta = new CyclicBarrier(3,
new Runnable() {
            @Override
            public void run() {
                System.out.println("Tutti presenti, possiamo
iniziare!");
            }
        });
        Festa festa = new Festa(cb);
        new Thread(festa,"Antonio").start();
        new Thread(festa,"Marcello").start();
        new Thread(festa,"Serena").start();
    }
}
```

L’output di quest’applicazione è il seguente:

```
Marcello è presente
Serena è presente
Antonio è presente
Tutti presenti, possiamo iniziare!
Antonio ha iniziato a festeggiare...
Marcello ha iniziato a festeggiare...
Serena ha iniziato a festeggiare...
```

Ma ovviamente l’ordine di arrivo dei partecipanti alla festa può cambiare ad ogni esecuzione.

Riepilogo

Abbiamo distinto le definizioni di **multithreading** e **multitasking**. Abbiamo gradualmente dato la definizione di thread che è piuttosto complessa (un processore virtuale che esegue codice su determinati dati). Abbiamo utilizzato la classe **`Thread`** e l’interfaccia **`Runnable`** per avere codice da far eseguire ai nuovi thread. Abbiamo parlato di **scheduler** e **priorità** ed abbiamo visto come questo concetto non sia quello determinante per gestire più thread contemporaneamente. È stato introdotto il modificatore **`volatile`**, che è applicabile solo alle variabili per indicare alla virtual machine di non creare copie di esse in memoria durante l’esecuzione. Abbiamo imparato a sincronizzare più thread introducendo i concetti di **monitor** e **lock** di un oggetto in due modi diversi: sincronizzando un metodo oppure un determinato blocco di codice. Abbiamo visto che è spesso necessario sincronizzare alcune parti di codice quando sono eseguibili da più thread. La sincronizzazione si limita a sincronizzare il codice, non i dati utilizzati. Abbiamo anche esplorato con un esempio la comunicazione tra thread, che si gestisce direttamente dal codice da eseguire in comune mediante i metodi **`wait()`**, **`notify()`** e **`notifyAll()`** della classe `Object`. Infine abbiamo accennato alle librerie e le tecnologie legate ai thread, ed in particolare al package **`java.util.concurrent`** e ai suoi sotto-package che introducono strutture di alto livello per la gestione avanzata dei thread.

Esercizi e altro materiale didattico sono disponibili all’indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper definire multithreading e multitasking (unità 15.1)	☐	
Comprendere la dimensione temporale introdotta dalla definizione dei thread in quanto oggetti (unità 15.2)	☐	
Saper creare ed utilizzare thread tramite la classe `Thread` e l'interfaccia `Runnable` (unità 15.2)	☐	
Definire che cos'è uno scheduler e i suoi comportamenti riguardo alle priorità dei thread (unità15.3)	☐	
Saper utilizzare i modificatori `volatile` e `synchronized` (unità 15.3, 15.4)	☐	
Far comunicare i thread (unità15.5)	☐	
Conoscere il nucleo della libreria sulla concorrenza (unità15.6)	☐	

Note:

16

Espressioni Lambda

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Comprendere la sintassi e le modalità d'uso delle espressioni lambda (unità 16.1).
- ✔ Saper decidere quando utilizzare un'espressione lambda o una classe anonima (unità 16.1).
- ✔ Saper gestire la dinamicità del passaggio di codice ad un metodo con un'espressione lambda (unità 16.1).
- ✔ Saper definire ed utilizzare tutte le quattro diverse sintassi dei reference a metodi (unità 16.2).
- ✔ Saper utilizzare le interfacce funzionali definite nel package `java.util.function` (unità 16.3).

Quando Java nacque nel 1995, era un linguaggio molto più semplice del linguaggio che stiamo studiando in queste pagine. Non esistevano generics, classi anonime, cicli foreach, import statici, varargs, enumerazioni, asserzioni, annotazioni e tante librerie. Negli anni sono stati introdotte gradualmente queste caratteristiche ad una ad una, ed il linguaggio è diventato sempre più potente e completo, ma anche più complicato da imparare. Soprattutto nella versione 5, il linguaggio subì una clamorosa

svolta con l'introduzione di tantissime novità. Quello che però non era mai cambiata sino ad ora, era la coerenza d'essere un linguaggio orientato agli oggetti. Ogni nuova funzionalità che veniva introdotta era sempre progettata per integrarsi nella filosofia object oriented. In 19 anni si è sempre cercato di far rimanere Java un linguaggio più ad oggetti possibile. Questo ha fatto di Java il linguaggio di riferimento per il paradigma ad oggetti. Tanto per fare un esempio, oggi negli atenei italiani si insegna Java (in molti casi usando questo libro) anche per spiegare gli oggetti. Nella didattica accademica altri linguaggi object oriented sono stati accantonati nel tempo, perché il supporto ai paradigmi non è così chiaro e definito come quello che offre Java. La coerenza di Java è sempre stata apprezzata, e la filosofia dell'Object Orientation si è affermata nel tempo. Java è stato considerato da molti il linguaggio ideale per progettare soluzioni ad oggetti, anche se lo scotto da pagare era una certa prolissità nel codice degli algoritmi.

Negli ultimi tempi però la scena della programmazione mondiale è cambiata. Con l'avvento di processori multi-core nell'uso domestico, la programmazione funzionale è stata rivalutata. Questo è successo un po' perché essa ben si adatta alle nuove architetture hardware, e un po' perché il suo accantonamento in favore dell'Object Orientation, fu dovuto soprattutto al suo utilizzo improprio. In fondo il paradigma funzionale ha dominato la scena della programmazione per decenni prima di passare in secondo piano negli anni novanta. Grazie anche alla novità portata da linguaggi moderni come Scala, Groovy e il framework Microsoft .Net, gli utenti Java hanno richiesto a gran voce l'aggiornamento del linguaggio, e l'introduzione di caratteristiche funzionali avanzate. Con questi linguaggi infatti è possibile scrivere algoritmi con un numero di righe nettamente inferiore rispetto a quello che poteva fare Java. Un processo lunghissimo, iniziato con le richieste dei programmatori e successive feroci battaglie nei forum su internet, hanno portato alla definizione di cosa sarebbe dovuto rientrare nelle nuove versioni di Java. Quando la versione 6 invece di rinnovare il linguaggio puntò soprattutto nella modernizzazione di alcune librerie, in molti rimasero delusi. La delusione salì ancora di più quando Oracle comprò Sun Microsystems e dopo ben cinque anni dalla release 6, pubblicò la versione 7 estromettendo le novità promesse più interessanti (tra cui proprio le espressioni lambda). Qualcuno aveva già definito Java un linguaggio morto, e lo slogan "Java is dead" lo si può ancora trovare oggi in molte

discussioni sul web. In effetti ci eravamo abituati a release ogni due anni con Sun, cosa che con Oracle non è più accaduta sino ad ora.

Da Java 9 in poi la filosofia di rilasci di Java cambierà! Ogni sei mesi verrà rilasciata una versione minore, che conterrà non solo le solite patch di sicurezza e correzioni di bug, ma anche tutte le novità che Oracle è riuscita a consolidare in tale periodo. In questo modo i tempi di sviluppo per l'integrazione delle nuove funzionalità saranno più brevi ed efficaci, visto che ne saranno introdotte per ogni minor release solo poche. Dopo due anni, con la stessa filosofia, sarà rilasciata una major release, a cui ci si riferisce solitamente come LTS release (che sta per "long term support release" ovvero "rilascio con supporto a lungo termine"). In questo modo i rilasci non saranno più rimandati nel caso accada che una certa novità tanto attesa non sia pronta per tempo (come è successo negli ultimi anni). Semplicemente ci sarà una nuova release che non avrà tale novità, ma che sarà implementata se possibile nella prossima minor release. Quanto detto comporta che oggi già sappiamo che a marzo 2018 sarà pubblicata una nuova minor version di Java, ne avremo un'altra nel settembre 2018, ancora un'altra nel marzo 2019 e a settembre 2019 avremo già Java 10!

Con l'avvento di Java 8 fu però apportata una vera e propria rivoluzione, e tale versione può essere considerata come la più innovativa in tutta la storia di Java. Con l'introduzione delle espressioni lambda e la possibilità di poter referenziare i metodi, la **filosofia funzionale** fa il suo ingresso nella programmazione Java! La nuova sfida è quella di far convivere i due paradigmi nelle nostre applicazioni, in modo tale da ottenere il meglio della programmazione. Ovviamente non sarà facile, ed avvertiamo il lettore (anche quello già esperto) che questo capitolo potrebbe risultare particolarmente ostico.

16.1 Espressioni lambda

Nel ventesimo secolo visse il famoso matematico Alonzo Church, docente di Princeton che ebbe tra l'altro studenti come Alan Turing. Church è ricordato soprattutto per la definizione del **Lambda Calcolo**, un sistema formale (ovvero una dimostrazione) sulla effettiva calcolabilità delle funzioni. La sua dimostrazione è alla base dei concetti di funzione e di ricorsività, strumenti essenziali della programmazione funzionale. Nella sua opera, Church usò l'undicesima lettera dell'alfabeto greco λ (Lambda) per evidenziare i parametri delle funzioni. Il nome *espressione Lambda* deriva quindi dal Lambda Calcolo da cui ebbe inizio in qualche modo la storia della programmazione.

Sembra che Church fu ispirato ad utilizzare la lettera λ da una delle opere fondamentali della matematica moderna: il "Principia Mathematica" di Alfred Whitehead e Bertrand Russel. In questo testo si usava il simbolo di accento ^, per evidenziare le variabili e questo ispirò Church ad utilizzare in primo luogo la lettera lambda maiuscola Λ, salvo poi ripiegare per qualche motivo sulla lettera minuscola λ.

Un**'espressione lambda** è detta anche **funzione anonima** (in inglese **anonymous function).** Questo perché in effetti si tratta proprio di una funzione, quindi non un metodo appartenente ad una classe e chiamato tramite un oggetto, ma una funzione senza nome definita al volo in maniera simile a come abbiamo fatto con le classi anonime. Per esempio, con la seguente classe anonima:

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Prima di Java 8: Classe
anonima");
    }
}).start();
```

abbiamo fatto partire un thread passandogli come oggetto `Runnable`, un'implementazione creata al volo con la sintassi di classe anonima, ridefinendo il metodo `run()`. Siamo abituati oramai a questo tipo di sintassi che sino ad ora ha rappresentato il modo più sintetico per fare un override al volo. Il problema è che se c'è solo un metodo di cui fare override, questo tipo di sintassi sembra comunque troppo verbosa. Con Java 8 possiamo riscrivere il frammento di codice precedente usando un'espressione lambda (evidenziata in grassetto) al posto della classe anonima:

```
new Thread(()->System.out.println("Java 8: Funzione
anonima")).start();
```

riducendo da cinque ad un'unica riga il nostro codice.

Le espressioni lambda sono anche note come "closures" (in italiano dovremmo tradurre "chiusure", ma è un termine che viene usato solitamente in inglese). Precisamente una closure è un'espressione lambda che fa uso di variabili che non sono parametri e non sono variabili locali al blocco di

codice che definisce l'espressione. Tecnicamente anche le classi anonime erano delle closure, ma le espressioni lambda sono closure più sintetiche e simili a quelle definite in altri linguaggi.

16.1.1 Sintassi

La sintassi di un'espressione lambda è forse la più complicata tra le sintassi di elementi definite da Java. Infatti è possibile sintetizzare questa sintassi omettendo tutto quanto il compilatore può dedurre da solo. In generale la sintassi è la seguente:

```
([lista di parametri]) -> {blocco di codice}
```

Il primo elemento è una lista di parametri. Nell'esempio precedente la lista di parametri era vuota, ed è stata esplicitata con un blocco di parentesi tonde vuote. Poi segue una "freccia", costituita dal segno di sottrazione "`-`", concatenato con il simbolo maggiore "`>`". L'ultima parte è costituita dal blocco di codice della funzione da eseguire. Si noti che nel nostro primo esempio abbiamo evitato di circondare il blocco di codice con delle parentesi graffe, visto che si tratta di un'unica istruzione. Saremmo stati obbligati ad usare le parentesi graffe nel caso il blocco di codice fosse stato costituito da più di una riga. Ma come è possibile scrivere un'espressione lambda passandola al costruttore della classe `Thread`? Come fa il compilatore a capire il significato di quello che stiamo scrivendo? Il segreto è che il costruttore di `Thread` accetta un'istanza di `Runnable` che è un'interfaccia funzionale.

Ricordiamo ancora una volta che un'interfaccia è detta funzionale quando ha un unico metodo definito astratto (ma potrebbe contenere anche tanti metodi di default implementati, metodi statici, metodi privati, costanti e così via). Questo metodo a volte è definito come metodo SAM (che sta per "Single Abstract Method" ovvero "Singolo Metodo Astratto").

L'interfaccia `Runnable` definisce il metodo `run()`, e con la sintassi dell'espressione lambda, il compilatore è capace di dedurre automaticamente che stiamo riscrivendo proprio quel metodo. Infatti `run()` è l'unico metodo astratto definito in `Runnable`. Quindi il compilatore ha dedotto automaticamente che al costruttore della classe `Thread` avremmo dovuto passare un'istanza dell'interfaccia funzionale `Runnable`, e che l'espressione lambda va a sostituire l'implementazione del suo unico metodo astratto `run()`. Quindi dovremmo parlare di doppia deduzione. L'unica cosa che può fare un'espressione lambda è sostituire l'implementazione di un'interfaccia funzionale, e forse questa è la sua definizione più corretta. Infatti il seguente statement è valido:

```
Runnable r = ()->System.out.println("Java 8: Funzione
anonima");
```

Abbiamo assegnato un reference `r` di tipo `Runnable` all'espressione lambda, il che ci porta a pensare ad essa, non più come a una funzione, ma come a un oggetto.

16.1.2 Comprendere le espressioni lambda

Quando si ha a che fare con le espressioni lambda, all'inizio si può rimanere spiazzati. Come abbiamo visto, la sintassi ci permette di evitare di scrivere parentesi tonde, tipi che si possono dedurre, comandi di tipo `return` e parentesi graffe. Questo perché questa nuova funzionalità vuole essere soprattutto un modo per abbassare la verbosità del linguaggio. Se da un lato scriviamo meno codice, dall'altro potremmo perdere in leggibilità. Il segreto per non perdere la bussola leggendo il codice di questo capitolo (e di quelli successivi) è quello di non dimenticare mai l'equivalenza tra classe anonima ed espressione lambda. Una volta fatto proprio il concetto che un'espressione lambda possa essere referenziata con un reference di un'interfaccia funzionale, e che la sua definizione non è altro che l'implementazione dell'unico metodo astratto (SAM) dell'interfaccia funzionale, siamo già ad un ottimo punto per una buona comprensione di questo argomento. Ripetiamo, l'unica cosa che può fare un'espressione lambda è sostituire un'interfaccia funzionale.

16.1.3 Espressione Lambda vs Classe Anonima

Abbiamo quindi appena asserito che il concetto fondamentale che è alla base delle espressioni lambda, è che queste hanno l'unico obiettivo di sostituire un'interfaccia funzionale. Abbiamo visto che questo però lo può fare anche una classe anonima (come abbiamo visto nel paragrafo 10.2), quindi ora cercheremo di capire le differenze tra i due approcci per trarne delle conclusioni.

16.1.3.1 Sinteticità

Il vantaggio principale nell'utilizzare un'espressione lambda in luogo di una classe anonima, risiede essenzialmente nella sinteticità dell'espressione. Nell'esplicitare un'espressione lambda è anche possibile omettere:

- le parentesi graffe del blocco di codice quando questo è costituito da un'unica riga, ed è anche possibile evitare il; alla fine dello statement;
- il tipo dei parametri quando non c'è possibilità di errore;
- le parentesi tonde che circondano la lista dei parametri, nel caso quest'ultima fosse costituita da un unico elemento;
- nel caso il blocco di codice preveda solo un'istruzione di tipo `return`, allora la parola `return` è superflua e non si deve proprio usare (pena un errore in compilazione).
- Quindi le espressioni lambda sono state progettate per essere il più possibile sintetiche, e permettere al programmatore di scrivere meno codice. Per esempio consideriamo l'interfaccia funzionale `ActionListener` del package `java.awt` (questo package contiene classi per realizzare interfacce grafiche dipendenti dal sistema operativo, cfr. appendice Q). Questa definisce un unico metodo astratto `actionPerformed()` che prende in input un oggetto di tipo `ActionEvent`. È possibile registrare (ovvero associare) per un certo elemento grafico (supponiamo un bottone `javax.swing.JButton`) un'implementazione di `ActionListener`, tramite il metodo `addActionListener()`. Dopo avere effettuato quest'associazione, ogni volta che il bottone sarà cliccato sarà automaticamente chiamato

dalla JVM il metodo `actionPerformed()`. Prima di Java 8 dovevamo scrivere, nella migliore delle ipotesi (ovvero con una classe anonima), il seguente codice:

```
JButton button1 = new JButton("Class Anonima");
button1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Prima di Java 8: Classe anonima
");
    }
});
```

Ora invece è possibile semplificare la sintassi in questo modo:

```
JButton button2 = new JButton("Espressione lambda");
button2.addActionListener(e->System.out.println("Java 8:
Funzione anonima"));
```

Si noti che abbiamo omesso sia il tipo (`ActionEvent`) del parametro `e`, sia le parentesi tonde che solitamente circondano la lista dei parametri, nonché il blocco di codice per il corpo del metodo e il `;` finale dello statement!

Alle interfacce grafiche create con le librerie Swing e AWT e la relativa gestione degli eventi è dedicata l'appendice on line Q. Il ventesimo capitolo di questo libro invece, è dedicato alla creazione di interfacce grafiche con il nuovo standard JavaFX.

16.1.3.2 Regole e visibilità

Un altro vantaggio che avremo utilizzando le espressioni lambda in luogo delle classi anonime, è quello di non avere più a che fare con le complesse leggi che regolano l'utilizzo di queste ultime (cfr. paragrafo 10.1.2). Per esempio, all'interno di una classe anonima, il reference `this` si riferisce all'oggetto corrente istanziato dalla classe anonima. Per riferirci ai membri della classe che include la classe anonima dobbiamo utilizzare una sintassi tipo:

```
NomeClasseEsterna.this.nomeMembro
```

Con le espressioni lambda invece, il reference `this` si riferisce direttamente alla classe in cui è inclusa l'espressione. Per esempio se eseguissimo la seguente classe:

```
public class LambdaThis {
     private String stringa = "variabile d'istanza della
classe";
     public void metodoContenenteLambda() {
         String stringa = "variabile locale del metodo
contenente";
         new Thread(()-
>System.out.println(this.stringa)).start();
         new Thread(()-
>System.out.println(stringa)).start();
     }
     public static void main(String args[]) {
         LambdaThis lambdaThis = new LambdaThis();
         lambdaThis.metodoContenenteLambda();
    }
}
```

sarebbe stampato in output:

```
variabile d'istanza della classe
variabile locale del metodo contenente
```

Per quanto riguarda le variabili locali e i parametri locali dichiarati all'interno di un'espressione lambda, esse condividono il loro scope (visibilità) con il blocco di codice dove sono definite. Questo significa che se riscrivessimo il metodo `metodoContenenteLambda()` del precedente esempio in questo modo:

```
public void metodoContenenteLambda() {
    String stringa = "variabile locale del metodo
contenente";
    new Thread(()-> {
        String stringa = "variabile locale nell'espressione
lambda";
        System.out.println(stringa);
    } ).start();
}
```

otterremmo il seguente errore in compilazione:

```
LambdaThis.java:8: error: variable stringa is already defined
in
  method metodoContenenteLambda()
            String stringa = "variabile locale
nell"espressione lambda";
                   ^
1 error
```

infatti è come se avessimo dichiarato due volte la variabile `stringa`.

Come per le classi anonime anche le espressioni lambda possono utilizzare variabili dichiarate al di fuori dell'espressione se e solo se queste sono dichiarate `final` o sono effettivamente non modificabili (cfr. paragrafo 10.1.2). Quindi il seguente frammento di codice:

```
 public void startCount() {
     int count = 0;
     new Thread(() -> {
         while (count < 100) {
             System.out.println(count++);
         }
     }).start();
 }
```

produrrà il seguente output in fase di compilazione:

```
TestCountArrayTrick.java:5: error: local variables referenced
from a
  lambda expression must be final or effectively final
            while (count < 100) {
                   ^
```

infatti abbiamo provato a modificare una variabile del metodo `startCount()` con l'operatore di post-incremento.

In molti casi il compilatore si accorgerà se proviamo a modificare una variabile dichiarata esternamente all'espressione lambda, ma non possiamo farvi affidamento completamente. Per esempio se la variabile in questione è di tipo reference, allora anche dichiarandola `final`, potremmo sempre modificarne la struttura interna senza che il compilatore se ne accorga. Infatti un famoso trucco per far funzionare l'esempio precedente, è impacchettare il valore di `count` mettendolo in un array (che è un oggetto) di dimensione uno, come nel seguente frammento di codice:

```
public void startCount() {
    int count = 0;
    int array[] = new int[1];
    array[0] = count;
    new Thread(() -> {
        while (array[0] < 100) {
            System.out.println(array[0]++);
        }
    }).start();
}
```

Ogni variabile d'istanza invece è sempre utilizzabile e modificabile.

16.1.3.3 Dinamicità

Nel paragrafo 10.2.1 abbiamo mostrato come utilizzare le classi anonime per rendere dinamico il nostro codice ed evitare i copia-incolla. In particolare abbiamo visto come poter *passare del codice* come parametro ad un metodo, che infine lo eseguirà. Ma le espressioni lambda sono l'ideale per poter passare codice! Riprendiamo quindi l'esempio che abbiamo descritto alla fine del paragrafo 10.2.1, dove avevamo la necessità di filtrare i titoli dei film di una videoteca secondo dei criteri di selezione (nel nostro caso i film dovevano avere un voto di recensione più alto di 3, ed essere film di fantascienza). Ricordiamo che avevamo creato la classe `Film`:

```
public class Film {
    private String nome;
    private String genere;
    private int mediaRecensioni;

    public Film (String nome, String genere, int
mediaRecensioni) {
        this.nome = nome;
        this.genere = genere;
        this.mediaRecensioni = mediaRecensioni;
    }
    // Metodi set e get omessi...
    public String toString(){
        return getNome();
    }
}
```

l’interfaccia (che ora possiamo anche annotare come interfaccia funzionale) `FiltroFilm`:

```
@FunctionalInterface
public interface FiltroFilm {
    boolean filtra(Film film);
}
```

e la classe `Videoteca`:

```
public class Videoteca {
    private Film[] films;

    public Videoteca() {
        films = new Film[10];
        caricaFilms();
    }

    public void setFilms(Film[] films) {
        this.films = films;
}
    public Film[] getFilms() {
        return films;
    }

    public Film[] getFilmFiltrati(FiltroFilm filtroFilm) {
        Film [] filmFiltrati = new Film[10];
        for (int i=0, j=0; i<10; i++) {
            if (filtroFilm.filtra(films[i])) {
                filmFiltrati[j] = films[i];
                j++;
            }
        }
        return filmFiltrati;
    }

    private void caricaFilms() {
         //Caricamento film...
    }
}
```

Infine avevamo utilizzato il metodo `getFilmFiltrati()` passando due diverse implementazioni di `FilmFiltro` usando la notazione di classe anonima nel seguente modo:

```
Videoteca videoteca = new Videoteca();
System.out.println("Bei Film:");
Film[] beiFilms = videoteca.getFilmFiltrati(new FiltroFilm()
{
    @Override
    public boolean filtra(Film film) {
         return film.getMediaRecensioni() > 3;
    }
});
System.out.println("\nFilm di Fantascienza:");
Film[] filmDiFantascienza = videoteca.getFilmFiltrati(new
FiltroFilm() {
    @Override
    public boolean filtra(Film film) {
        return "Fantascienza"equals(film.getGenere());
    }
});
```

Ora che conosciamo la sintassi delle espressioni lambda, possiamo riscrivere il precedente frammento di codice in maniera più compatta:

```
Videoteca videoteca = new Videoteca();
System.out.println("Bei Film:");
Film[] beiFilms = videoteca.getFilmFiltrati((Film film)->
  film.getMediaRecensioni() > 3);
System.out.println("\nFilm di Fantascienza:");
Film[] filmDiFantascienza = videoteca.getFilmFiltrati((Film
film)->
  "Fantascienza".equals(film.getGenere()));
```

Si noti che in un metodo normale, nel blocco di codice del metodo avremmo usato anche un comando `return` come abbiamo fatto quando abbiamo creato la classe anonima precedente. Invece la natura sintetica della sintassi delle espressioni lambda richiede che il comando `return` sia omesso.

Volendo possiamo anche creare altri filtri al volo in maniera molto semplice. Per esempio di seguito filtriamo i film che hanno un nome che termina con la lettera “s”:

```
Film[] filmCheFinisconoConS =
videoteca.getFilmFiltrati((Film film)->
  film.getNome().endsWith("s"));
```

Anche se tutto questo si poteva già fare con le classi anonime, sembra

evidente che le espressioni lambda siano decisamente più adeguate a risolvere questo tipo di problemi.

16.1.3.4 Gestione delle eccezioni

Quando utilizziamo le espressioni lambda, dobbiamo fare attenzione alle cosiddette checked exception, ovvero alle eccezioni che non sono sottoclassi di `RuntimeException`. Per tale motivo dovrebbero essere dichiarate nella clausola `throws` della dichiarazione del metodo che potrebbe lanciare l'eccezione, nel caso non vengano gestite (cfr. capitolo 9). Ma le espressioni lambda non dichiarano metodi, e non ci sarebbe spazio per utilizzare una clausola `throws` se non nel metodo dell'interfaccia funzionale che rappresentano. E questo non è sempre possibile. Cerchiamo di capire meglio con un semplice esempio. Il seguente frammento di codice:

```
 new Thread(()-> {
     Thread.sleep(1000);
     System.out.println("Hello World");
 } ).start();
```

darà luogo ad un errore in compilazione:

```
LambdaException.java:4: error: unreported exception
  InterruptedException; must be caught or declared to be
thrown
            Thread.sleep(1000);
                         ^
1 error
```

abbiamo due possibilità per risolvere il problema, o gestire l'eccezione con un `try-catch`:

```
 new Thread(()-> {
     try {
         Thread.sleep(1000);
     }
     catch (InterruptedException exc) {
         exc.printStackTrace();
     }
     System.out.println("Hello World");
 } ).start();
```

o utilizzare un'interfaccia funzionale che definisca il metodo astratto con la clausola `throws` richiesta. Nel nostro caso non possiamo farlo con l'interfaccia `Runnable`, perché la definizione del metodo `run()` non dichiara una clausola `throws`, e per le proprietà dell'override, non possiamo definire una sottointerfaccia che aggiunga la clausola `throws` (cfr. paragrafo 9.5.2). Dobbiamo ripiegare quindi su un meccanismo un po' più complesso: usare l'interfaccia (funzionale) `Callable` e il suo metodo `call()` che invece dichiara una clausola `throws` per `Exception`, e passarlo in input ad un metodo `submit()` di un `ExecutorService` (cfr. paragrafo 15.6.6.3), invece di lanciare il comando `start()` di un `Thread` come nell'esempio precedente. Potremmo per esempio creare la seguente interfaccia funzionale:

```
@FunctionalInterface
interface MyCallable extends Callable<Void> {
    @Override
    Void call() throws InterruptedException;
}
```

Con la quale abbiamo esteso l'interfaccia generica `Callable` passandogli il tipo `Void`. Questo implicherà che il metodo `call()` ritornerà un tipo `Void`.

> **Un oggetto `Void` è come se fosse una classe wrapper per il valore di ritorno `void` dei metodi.**

Poi abbiamo riscritto il metodo `call()` in modo tale che dichiari nella sua clausola `throws` la `InterruptedException` che l'espressione lambda dovrà gestire. Quindi ora possiamo riscrivere il frammento di codice contenente l'espressione lambda in questo modo:

```
MyCallable callable = ()-> {
     Thread.sleep(1000);
     System.out.println("Hello World");
     return null; // necessario perché Void è un classe
};
ExecutorService pool = Executors.newFixedThreadPool(1);
pool.submit(callable);
pool.shutdown();
```

Abbiamo assegnato l'espressione lambda ad un reference della nostra estensione `MyCallable`. Si noti che siamo stati costretti a ritornare `null`, perché comunque il metodo si aspetta un tipo di ritorno `Void`. Queste righe quindi ora compileranno perché il metodo `call()` che stiamo definendo con l'espressione lambda definisce la clausola `throws` richiesta. Poi istanziamo un `ExecutorService pool` di dimensione `1`, e chiamiamo il metodo `submit()` passandogli l'espressione lambda appena definita. Infine chiudiamo l'executor `pool`. Sicuramente queste situazioni non si adattano molto bene alla natura sintattica delle espressioni lambda.

Si noti che ci saremmo pure potuti risparmiare la definizione dell'interfaccia `MyCallable`, ed avremmo potuto scrivere direttamente:

```
Callable<Void> callable = ()-> {
    Thread.sleep(1000);
    System.out.println("Hello World");
    return null;
};
```

visto che il metodo `call()` di `Callable` ha una clausola `throws` generica (`Exception`). In questo modo però, altre eccezioni checked diverse da `InterruptedException` non sarebbero state più sollevate dal compilatore, nel caso ce ne fosse stato bisogno.

Il supporto delle espressioni lambda ai debugger purtroppo non è impeccabile. Infatti questi devono fare i conti, non solo con le eventuali eccezioni lanciate, ma anche con le varie "licenze di taglio" concesse alle espressioni lambda (tipi, nomi, parentesi, punti e virgola e così via). Può capitare quindi che debuggare un'espressione lambda possa diventare complicato anche utlizzando un IDE evoluto.

16.1.3.5 Quando usare le espressioni lambda

Dovremmo usare le espressioni lambda quando il nostro obiettivo è quello di passare in maniera dinamica un certo algoritmo ad un altro metodo. Questo serve per eseguire l'algoritmo in un contesto definito dal metodo a cui stiamo

passando l'algoritmo. Passiamo del codice ad un metodo, per dare a quest'ultimo la responsabilità di chiamarlo al momento giusto. Un buon esempio è quando abbiamo passato un'espressione lambda al metodo `addActionListener()`, allo scopo che sia eseguita alla pressione di un bottone su un'interfaccia grafica. Potremmo anche passare del codice ad un metodo per farlo eseguire in base ad un altro evento come il lancio di un'eccezione. Oppure potremmo anche fare in modo che il nostro codice sia eseguito mentre si itera su ogni elemento di un array. In generale passare una espressione lambda, significherà far decidere al metodo a cui abbiamo passato il codice, il momento in cui eseguirlo. Inoltre dobbiamo pensare sempre al fatto che usare un'espressione lambda significa sostituire un'interfaccia funzionale in maniera sintetica. Nel caso invece il nostro obiettivo sia quello di estendere le funzionalità o aggiungere dati all'astrazione offerta di un'interfaccia funzionale, allora sarebbe preferibile utilizzare una classe anonima, o come vedremo a partire dal prossimo paragrafo, un reference a metodo.

16.2 Reference a metodi

A questo punto sappiamo che abbiamo la possibilità di chiamare **funzioni anonime.** Le dovremmo preferire sicuramente alle classi anonime nel caso il loro blocco di codice contenga pochi statement. Molto spesso però, esse contengono semplicemente un'unica chiamata ad un metodo. In casi come questo, esiste una sintassi ancora più compatta delle espressioni lambda: quella dei reference a metodi esistenti. In questo caso parliamo di **reference a metodi** (in inglese **method references**). Riprendiamo l'esempio della videoteca di prima. Se creassimo questa classe che definisce dei filtri:

```
public class Filtri {
    public static boolean isBelFilm(Film film) {
        return film.getMediaRecensioni() > 3;
    }

    public static boolean isFilmDiFantascienza(Film film) {
        return "Fantascienza".equals(film.getGenere());
    }
}
```

potremmo passare l'implementazione dei metodi usando la sintassi dei *reference ai metodi* nel seguente modo:

```
Film[] filmDiFantascienza =
  videoteca.getFilmFiltrati(Filtri::isFilmDiFantascienza);
stampaFilm(filmDiFantascienza);
```

L'effetto è il medesimo delle espressioni lambda, ma la sintassi è ancora più semplice e compatta. Infatti in questo caso abbiamo specificato solo il nome della classe (`Filtri`) il simbolo `::` e il nome del metodo (`isFilmDiFantascienza`) senza neanche specificare le parentesi (ed eventualmente i parametri). Attenzione che non stiamo semplicemente chiamando il metodo `isFilmDiFantascienza()`, altrimenti avremmo direttamente scritto:

```
Film[] filmDiFantascienza =
  videoteca.getFilmFiltrati(Filtri.isFilmDiFantascienza());
```

che però non avrebbe compilato in quanto `isFilmDiFantascienza()` ritorna un booleano e non possiamo passarlo al metodo `getFilmFiltrati()`. Infatti quest'ultimo si aspetta in input un oggetto di tipo `FiltroFilm`. Insomma, con le espressioni lambda sostituivamo un'istanza di un'interfaccia funzionale, passandogli l'implementazione dell'unico metodo definito. Con la sintassi del reference a metodo, andiamo a sostituire un'istanza di un'interfaccia funzionale, passandogli il nome di un metodo già definito, che corrisponde per tipo di ritorno e parametri in input, all'unico metodo SAM definito nell'interfaccia funzionale. Anche in questo caso la deduzione automatica dei tipi fa tutto il lavoro per noi.

Inoltre rispetto ad una espressione lambda questo tipo di sintassi si fa preferire nel caso abbiamo bisogno di utilizzare più volte la chiamata al reference di un metodo. Infatti con una espressione lambda spesso il vantaggio è quello di creare del codice al volo. Ma se questo codice dobbiamo usarlo più volte in classi diverse probabilmente inizieremmo a far uso dei copia incolla. E come abbiamo già asserito in precedenza, il copia-incolla è una pratica estremamente sconveniente. Inoltre le espressioni lambda si prestano bene a definire implementazioni costituite possibilmente da un'unica riga. I reference a metodi invece, essendo dichiarati in un proprio metodo, possono tranquillamente essere utilizzati in ogni occasione in cui si

debba passare del codice dinamicamente ad un metodo.

16.2.1 Sintassi

La sintassi dei reference a metodi può essere di quattro tipi.

- **Reference a un metodo statico** (in inglese **reference to a static method**):

```
NomeClasse::nomeDelMetodoStatico
```

- **Reference a un metodo d'istanza** (in inglese **reference to an instance method of a particular object**):

```
NomeOggetto::nomeDelMetodoDIstanza
```

- **Reference a un metodo d'istanza di un certo tipo** (in inglese **reference to an instance method of an arbitrary object of a particular type**):

```
NomeTipo::nomeDelMetodoDIstanza
```

- **Reference a un costruttore** (in inglese **reference to a constructor**):

```
NomeClasse::new
```

Cercheremo di capire in dettaglio le differenze tra queste quattro tipologie di reference a metodo nei prossimi paragrafi.

16.2.2 Reference a un metodo statico

Abbiamo appena visto un esempio di **reference a metodo statico** con la classe `Filtri`. In particolare abbiamo già notato come questo tipo di reference a metodo sia del tutto equivalente ad un'espressione lambda, tanto che l'abbiamo usato proprio allo stesso modo. La differenza risiede nel fatto che un reference ad un metodo, usa un metodo già creato e non si crea al volo la sua implementazione come con le espressioni lambda. Nel caso esistesse la classe `Filtri`, avremmo comunque potuto usare un'espressione lambda con la seguente sintassi:

```
Film[] filmDiFantascienza = videoteca.getFilmFiltrati(
  (Film film)->Filtri.isFilmDiFantascienza(film));
```

ma è più semplice e compatto usare un reference a metodo come abbiamo visto prima:

```
Film[] filmDiFantascienza =
  videoteca.getFilmFiltrati(Filtri::isFilmDiFantascienza);
```

16.2.3 Reference a un metodo d'istanza

Lo stesso discorso fatto per un reference a metodo statico può essere ripetuto per i **reference a metodi d'istanza.** L'unica differenza è che il metodo coinvolto non è statico, e che quindi bisogna usare un oggetto istanziato dalla classe che definisce il metodo per utilizzare il reference al metodo. Per esempio, creiamo una classe che fornisca dei metodi per ordinare i film:

```
public class OrdinamentoFilm {
    public int ordinaPerNome(Film film1, Film film2) {
        return film1.getNome().compareTo(film2.getNome());
    }

    public int ordinaPerMediaRecensioni(Film film1, Film
film2) {
        Integer mediaRecensioniFilm1 =
           new Integer(film1.getMediaRecensioni());
        Integer mediaRecensioniFilm2 =
           new Integer(film2.getMediaRecensioni());
        return
mediaRecensioniFilm2.compareTo(mediaRecensioniFilm1);
    }
}
```

Per gestire l'ordinamento, abbiamo creato dei metodi che sfruttano il metodo `compareTo()` di `String` (ereditati dell'interfaccia funzionale `java.lang.Comparable`, che abbiamo approfondito nel paragrafo 12.8.1) ed `Integer`. Questa classe ci offre due tipi di ordinamento: alfabetico per il nome, e per media recensioni dal voto più alto al più basso. Possiamo quindi ordinare i film nel seguente modo:

```
Film[] films = videoteca.getFilms();
System.out.println(
  "\nFilm ordinati per nome con un instance method
reference: ");
OrdinamentoFilm ordinamentoFilm = new OrdinamentoFilm();
```

```
Arrays.sort(films, ordinamentoFilm::ordinaPerNome);
stampaFilm(films);
System.out.println(
  "\nFilm ordinati per media recensioni con un instance
method reference: ");
Arrays.sort(films,
ordinamentoFilm::ordinaPerMediaRecensioni);
stampaFilm(films);
```

È possibile fare questo perché indipendentemente dal nome del metodo che stiamo passando, il metodo `sort()` vuole in input il blocco di codice che deve rappresentare l'implementazione del metodo `compare()` della classe `Comparator`. Il compilatore trovando corrispondenza tra parametri in input e output permette questa sintassi grazie alla deduzione automatica. Anche questa volta quindi potevamo usare un'espressione lambda invece di un reference a metodo nel seguente modo:

```
Arrays.sort(films, (Film film1, Film film2)->
  film1.getNome().compareTo(film2.getNome()));
```

ma il reference a metodo è risultato ancora una volta più compatto.

È anche possibile usare i reference `this` e `super` usando questa sintassi. Infatti è possibile usare questi reference anche nelle espressioni lambda equivalenti. Per esempio se avessimo la classe:

```
public class Saluto {
    public void saluta() {
        System.out.println("Ciao");
    }
}
```

Potremmo estenderla con la seguente ed utilizzare un reference a metodo sfruttando `super`:

```
public class SalutoAsincrono extends Saluto {
    @Override
    public void saluta() {
        new Thread(super::saluta).start();
    }
}
```

16.2.4 Reference a un metodo d'istanza di un certo tipo

Questa **tipologia di sintassi** all'inizio può portare facilmente a confondersi, perché dopo aver visto i due precedenti tipi di reference a metodo, questo può sembrare un po' un mix tra i due. Infatti sembra di chiamare un metodo non statico con il nome di una classe, ma non è così che bisogna intendere la sintassi:

```
NomeTipo::nomeDelMetodoDIstanza
```

In altre parole, se abbiamo un metodo d'istanza da cui il compilatore può dedurre parametri di input e di output è possibile usare il nome il nome del tipo invece del nome di un oggetto. Per esempio con le seguenti istruzioni:

```
List<String> filmNames = new ArrayList<String>();
for(Film film:films) {
    filmNames.add(film.getNome());
}
Collections.sort(filmNames, String::compareToIgnoreCase);
stampaFilm(films);
```

abbiamo creato una lista `filmNames` di stringhe contenente i nomi dei film dell'esempio precedente. Poi abbiamo ordinato la collezione passando il reference del metodo di `compareToIgnoreCase()` che si trova nella classe stringa, che ordina le stringhe ignorando se sono composte da caratteri maiuscoli o minuscoli. Abbiamo usato però una sintassi del tipo:

```
NomeDellaClasse::nomeDelMetodoDIstanza
```

Si noti che il metodo `compareToIgnoreCase()` non è statico, quindi è una sintassi diversa dal primo caso che abbiamo visto. Infatti il metodo d'istanza `compareToIgnoreCase()` è definito così:

```
public int compareToIgnoreCase(String str) { //...
```

Quindi se dovessimo usare questo metodo normalmente con delle stringhe, potremmo usarlo in questo modo:

```
String s1 = "Rosalia";
String s2 = "rosalia";
System.out.println("\nRosalia vs rosalia = " +
s1.compareToIgnoreCase(s2));
```

Se dovessimo scrivere quindi l'equivalente istruzione con una espressione lambda, dovremmo scrivere:

```
Collections.sort(filmNames, (s1,s2)-
>s1.compareToIgnoreCase(s2));
```

Con l'espressione lambda abbiamo potuto evitare di scrivere i tipi di `s1` e `s2` (che potevamo chiamare anche `x` ed `y`, sono solo degli identificatori) perché il compilatore dal contesto ha potuto dedurre che si trattava di stringhe. Con la sintassi del reference a metodo che invece abbiamo usato nell'esempio specificando `String::compareToIgnoreCase`, abbiamo potuto evitare di specificare l'implementazione del metodo, e di conseguenza, specificare `s1` e `s2`, non avrebbe avuto senso, perché il compilatore può dedurlo automaticamente. Però abbiamo dovuto specificare `String`, per indicare al compilatore che ci stavamo riferendo al metodo di `String`, e non ad un altro metodo in un'altra classe. A quel punto il compilatore ha potuto dedurre che nel metodo `sort()` potrà usare gli elementi della collezione `filmNames` con l'implementazione di `compareToIgnoreCase()` definito nella classe `String`. Insomma, è stato necessario specificare solo il tipo da cui prendere il metodo specificato, senza necessità di specificare oggetti di tipo stringa perché non dobbiamo definire al volo un'implementazione del metodo che già esiste.

> **Vedremo più avanti, quando parleremo di stream nel prossimo capitolo, che ci saranno casi in cui non sarà disponibile un reference di un oggetto (perché questo non viene definito) all'interno delle cosiddette pipeline (cfr. paragrafo 17.8.2). Spesso all'interno delle pipeline questa sintassi sarà la scelta più logica.**

16.2.5 Reference a un costruttore

È possibile anche referenziare un costruttore. È molto semplice, basta usare la parola chiave `new` come nome del metodo da chiamare. In effetti un costruttore è come un metodo che ritorna un oggetto della sua classe, ecco perché assegnargli un reference ha senso. Passiamo come al solito a fare un

esempio usando una **factory**, ovvero una classe che ha il compito di creare oggetti (definizione semplificata di uso comune del design pattern **Factory Method).** Supponiamo di avere una semplice classe `Chitarra` come la seguente:

```
public class Chitarra {
    private String marca;

    public Chitarra (String marca) {
        this.marca = marca;
        System.out.println("Creata chitarra: " + marca);
    }
}
```

Come interfaccia funzionale invece, consideriamo la seguente:

```
@FunctionalInterface
public interface FabbricaChitarra {
    Chitarra getChitarra(String marca);
}
```

In una situazione pre-Java 8, dovremmo creare un'implementazione dell'interfaccia:

```
public class FabbricaChitarraImpl implements
FabbricaChitarra {
    @Override
    public Chitarra getChitarra(String marca) {
        return new Chitarra(marca);
    }
}
```

per poi usarla con un codice come il seguente:

```
FabbricaChitarra fabbricaChitarra = new
FabbricaChitarraImpl;
Chitarra chitarra1 = fabbricaChitarra.getChitarra("Fender");
```

Da Java 8 possiamo completamente evitare di creare la classe factory `FabbricaChitarraImpl` utilizzando un reference a costruttore:

```
FabbricaChitarra fabbricaChitarra = Chitarra::new;
Chitarra chitarra1 = fabbricaChitarra.getChitarra("Fender");
```

Anche in questo caso è possibile utilizzare una espressione lambda in luogo del reference al costruttore nel seguente modo:

```
FabbricaChitarra fabbricaChitarra =
  (marca) -> { return new Chitarra(marca); };
Chitarra chitarra1 = fabbricaChitarra.getChitarra("Fender");
```

Come è evidente, anche in questo caso è preferibile la sintassi del reference a costruttore.

16.3 Le interfacce funzionali del package `java.util.function`

Abbiamo visto come le espressioni lambda e i reference a metodi si utilizzino per implementare comportamenti definiti in interfacce funzionali. A parte quando abbiamo fatto esempi con `ActionListener` e `Runnable`, sino ad ora abbiamo creato noi le interfacce funzionali che ci servivano. Esistono però anche altre interfacce funzionali che possiamo usare, come `Iterable` con il suo metodo `iterator()` ed `Executor` con il suo metodo `execute()`. Java 8 ha introdotto il package `java.util.function`, che definisce una serie di interfacce funzionali che dovrebbero coprire quasi sempre i nostri bisogni. Quindi creeremo solo raramente delle interfacce funzionali a mano. Di seguito illustriamo le interfacce funzionali più importanti, che sono dichiarate tutte generiche. Il lettore potrà trovare altre decine di interfacce funzionali all'interno di questo package.

16.3.1 `Predicate`

L'interfaccia funzionale `Predicate` è dichiarata nel seguente modo (senza metodi di default, metodi statici, import, package e commenti):

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Come si può intuire dall'unico metodo astratto, è un'interfaccia perfetta per fare dei test che ritornano un booleano. Il tipo parametro `T` ci garantisce la

massima flessibilità sul parametro in input che può essere deciso al volo. Se riprendiamo ancora l'esempio della videoteca, potremmo eliminare del tutto l'interfaccia `FiltroFilm`, e usare al suo posto `Predicate`. Quindi dovremmo riscrivere solo il metodo `getFilmFiltrati()` che avevamo definito nella classe `Videoteca` nel seguente modo:

```
public Film[] getFilmFiltrati(Predicate<Film> filtroFilm) {
    Film [] filmFiltrati = new Film[10];
    for (int i=0, j=0; i<10; i++) {
        if (filtroFilm.test(films[i])) {
           filmFiltrati[j] = films[i];
           j++;
        }
    }
    return filmFiltrati;
}
```

Si noti che abbiamo solo dovuto usare come tipo parametro per `Predicate` la classe `Film`, e sostituire la chiamata al metodo `getFilmFiltrati()` con il metodo `test()`. Per il resto non è cambiato niente. `Predicate` è l'interfaccia funzionale ideale per filtrare oggetti, anche perché definisce ulteriori metodi per concatenare filtri: `or()` e `and()`. Quindi se volessimo filtrare film di fantascienza che hanno anche una buona media di recensioni, basterebbe scrivere:

```
System.out.println("\nBei film di Fantascienza:");
Predicate<Film> predicateFilmFantascienza =
  (Film film)->"Fantascienza".equals(film.getGenere());
Predicate<Film> predicateBeiFilm =
  (Film film)->film.getMediaRecensioni()>3;
Film[] filmDiFantascienzaBelli =
  videoteca.getFilmFiltrati(predicateFilmFantascienza.and(predi

stampaFilm(filmDiFantascienzaBelli);
```

Ed è evidente che ci sia una grande flessibilità!

16.3.2 `Consumer`

L'interfaccia funzionale `Consumer` è dichiarata nel seguente modo (senza metodi di default, metodi statici, import, package e commenti):

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

Essa è intesa per fornire funzionalità di aggiornamento dello stato interno dell'oggetto che gli viene passato, o comunque di usarlo (per esempio stamparne il metodo `toString()`). Come si può notare infatti, il metodo `accept()` non ha tipo di ritorno. Rimarchiamo ancora il fatto che sia un'interfaccia generica e quindi flessibile ad accettare qualsiasi tipo di oggetto. Questa è una caratteristica di tutte le interfacce funzionali del package `java.util.function`. Supponiamo di voler creare una classe che ci consenta di aggiornare al momento opportuno lo stato interno degli oggetti di tipo `Film`. Potremmo creare la seguente classe:

```
import java.util.function.Consumer;
public class SocialBacheca {
    public void aggiornaFilm(Film film, Consumer<Film>
consumer) {
        consumer.accept(film);
    }
}
```

Quindi se volessimo aggiornare la media voto delle recensioni di un film potremmo scrivere semplicemente:

```
Videoteca videoteca = new Videoteca();
SocialBacheca bacheca = new SocialBacheca();
Film films[] = videoteca.getFilms();
Film lanternaVerde = films[7];
bacheca.aggiornaFilm(lanternaVerde, film ->
film.setMediaRecensioni(5));
System.out.printf("La media voto delle recensioni di %s è
stata " +
  "aggiornata a %s", lanternaVerde.getNome(),
  lanternaVerde.getMediaRecensioni());
```

Si noti che nella espressione lambda, abbiamo usato il reference `film` (ma l'avremmo anche potuto chiamare `x`), che il compilatore riesce a dedurre essere di tipo `Film`. Anche nel caso di `Consumer`, possiamo sfruttare il suo metodo di default `andThen()` per concatenare aggiornamenti.

Per esempio con il seguente frammento di codice:

```
Film starWarsEpisodio1 = films[1];
Consumer<Film> aggiornaVotoConsumer = film ->
film.setMediaRecensioni(3);
Consumer<Film> aggiornaNomeConsumer =
  film -> film.setNome("Star Wars Episodio" 1 - La minaccia
fantasma");
bacheca.aggiornaFilm(starWarsEpisodio1,
  aggiornaNomeConsumer.andThen(aggiornaVotoConsumer));
System.out.printf("\nLa media voto delle recensioni di %s è
stata " +
  "aggiornata a %s", starWarsEpisodio1.getNome(),
  starWarsEpisodio1.getMediaRecensioni());
```

Aggiorneremo sia il nome sia la media voto di un film. Ma l'aggiornamento del voto avverrà solo se andrà a buon fine l'aggiornamento del nome.

16.3.3 `Supplier`

L'interfaccia funzionale `Supplier` è dichiarata nel seguente modo (senza import, package e commenti):

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Essa si adatta bene nei contesti in cui serve gestire una factory. Il suo metodo `get()` restituisce un istanza del tipo parametro dichiatato, e quindi è facile pensare alla sua implementazione per creare e restituire un oggetto. Riprendiamo quindi l'esempio della classe `Chitarra` usato per mostrare i reference a costruttori. Aggiungiamo il metodo `suona()` alla classe `Chitarra`:

```
public class Chitarra {
    private String marca;

    public Chitarra (String marca) {
        this.marca = marca;
        System.out.println("Creata chitarra: " + marca);
    }

    public void suona() {
```

```
        System.out.printf("Sta suonando una %s ...", marca);
    }
}
```

Poi creiamo la classe `Chitarrista` nel seguente modo:

```
import java.util.function.Supplier;
public class Chitarrista {
    public void suonaChitarra(Supplier<Chitarra>
marcaSupplier) {
        Chitarra chitarra = marcaSupplier.get();
        chitarra.suona();
    }
}
```

Si noti come il metodo `get()` restituisca un'istanza del tipo parametro `Chitarra`. Infine con il seguente frammento di codice, potremo far suonare al chitarrista la chitarra:

```
Chitarrista chitarrista = new Chitarrista();
Supplier<Chitarra> chitarraSupplier = ()-> new
Chitarra("Ibanez");
chitarrista.suonaChitarra(chitarraSupplier);
```

16.3.4 `Function`

L'interfaccia funzionale `Function` è dichiarata nel seguente modo (senza metodi di default, metodi statici, import, package e commenti):

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Quest'interfaccia ci permette di astrarre proprio il concetto di funzione più classico, dove c'è un input (impersonificato dal tipo parametro `T`) e un output (tipo parametro `R`). Esiste anche l'interfaccia funzionale `BiFunction` che è definita come segue (senza metodi di default, import, package e commenti):

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

dove gli input sono due (`T` ed `U`).

> **Esistono le equivalenti "Bi" interfacce anche per le altre interfacce funzionali** (`BiPredicate, BiSupplier` **e così via).**

Per l'esempio useremo `BiFunction`. Supponiamo di voler creare un metodo che esegue una funzione di trasformazione di due stringhe in un'unica stringa. Essa deve quindi prendere in input due stringhe e ritornarne una. Potremmo creare il seguente metodo:

```
public static String trasformaStringhe(String s1, String s2,
  BiFunction<String, String, String> biFunction) {
    return biFunction.apply(s1,s2);
}
```

Se volessimo sfruttare il metodo precedente per eseguire una concatenazione di due stringhe separandole con un trattino, potremmo scrivere:

```
BiFunction<String,String,String> concatenazioneConTrattino =
  (String s1, String s2) -> { return s1 + "-" + s2; };
System.out.println(
  trasformaStringhe("Andrea", "Simone",
concatenazioneConTrattino));
```

Si noti che nella libreria non esistono altre interfacce come `TriPredicate`, `TriFunction` e così via, ma all'occorrenza possiamo crearle in modo molto semplice. Ecco come definire una `TriFunction`:

```
@FunctionalInterface
public interface TriFunction<T, U, V, R> {
    R apply(T t, U u, V v);
}
```

16.3.5 `UnaryOperator` e composizione di espressioni lambda

L'interfaccia funzionale `UnaryOperator` è dichiarata nel seguente modo (senza commenti e metodi statici):

```
@FunctionalInterface
    public interface UnaryOperator<T> extends Function<T, T>
{
}
```

Come è possibile notare, `UnaryOperator` è semplicemente un'estensione dell'interfaccia funzionale `Function` e quindi eredita da essa il metodo astratto:

```
T apply(T t);
```

`UnaryOperator` astrae un'operazione che viene fatta su un singolo parametro e che produce un risultato dello stesso tipo del parametro. Essa quindi si adatta bene ad operazioni di tipo *trasformazione*. Quest'interfaccia ci sarà utile quando avremo intenzione di combinare più espressioni lambda per eseguirle in un solo colpo, evitando trasformazioni intermedie. Per esempio consideriamo la classe `Film` che abbiamo definito nel paragrafo 16.1.3.3. Creiamo una classe che astragga il concetto di programmazione del film:

```
public class ProgrammazioneFilm {
    private Film film;
    private LocalDate data;
    private LocalTime orario;

    public ProgrammazioneFilm(Film film, LocalDate data,
LocalTime orario) {
        this.film = film;
        this.data = data;
        this.orario = orario;
    }

    //Metodi setter e getter omessi

    public String toString() {
        return film.getNome() + " verrà proiettato il " +
getData() +
          " alle " + getOrario();
    }
}
```

Una classe `Cinema` avrà la responsabilità della schedulazione dei film:

```
public class Cinema {
```

```
    public static UnaryOperator<ProgrammazioneFilm>
componiCambi(
            UnaryOperator<ProgrammazioneFilm> op1,
            UnaryOperator<ProgrammazioneFilm> op2) {
        return x->op2.apply(op1.apply(x));
    }

    public static ProgrammazioneFilm cambiaProgrammazione(
      ProgrammazioneFilm pf,
UnaryOperator<ProgrammazioneFilm> op) {
        return op.apply(pf);
    }
}
```

Abbiamo definito al suo interno il metodo `componiCambi()`, che compone un'operazione del primo parametro `UnaryOperator op1` con il secondo parametro `op2`, per restituire sempre un oggetto `UnaryOperator`. Quindi questo metodo restituisce la composizione dei due `UnaryOperator` presi in input sotto forma sempre di uno `UnaryOperator`. Lo scopo è poi di riutilizzare l'oggetto composito per passarlo al metodo `cambiaProgrammazione()`.

È il primo esempio in cui vediamo come una espressione lambda viene ritornata come output di un metodo, ma questo non deve meravigliarci. Le espressioni lambda possono avere un reference, e quindi alla pari dei reference possono essere sia input che output di metodi.

Quest'ultimo infatti prende in input gli oggetti `ProgrammazioneFilm pf` e `UnaryOperator op`, e chiama il metodo `apply()` su `op`, passandogli `pf`. Il metodo ritornerà l'oggetto `pf` modificato con l'implementazione di `op`. Per comprendere meglio l'esempio, completiamolo con l'implementazione di un frammento di codice estratto da un ipotetico metodo `main()`:

```
UnaryOperator<ProgrammazioneFilm> ritardaDiUnOra = p-> {
    LocalTime orario = p.getOrario();
    orario = orario.plus(1, ChronoUnit.HOURS);
```

```
    p.setOrario(orario);
    return p;
};
UnaryOperator<ProgrammazioneFilm> ritardaDiUnGiorno = p-> {
    p.setData(p.getData().plus(1, ChronoUnit.DAYS));
    return p;
};
Film shining = new Film("Shining", "Thriller", 5);
ProgrammazioneFilm programmazioneShining = new
ProgrammazioneFilm(
  shining, LocalDate.of(2018, 5, 29), LocalTime.of(21,0));
UnaryOperator<ProgrammazioneFilm> ritardaDiUnOraEUnGiorno =
  Cinema.componiCambi(ritardaDiUnOra, ritardaDiUnGiorno);
System.out.println(Cinema.cambiaProgrammazione(
  programmazioneShining, ritardaDiUnOraEUnGiorno));
```

Abbiamo definito due `UnaryOperator` implementandoli con due semplici espressioni lambda. La prima (che definisce `ritardaDiUnOra`) incrementa l'orario di un oggetto `ProgrammazioneFilm` di un'ora. La seconda (che definisce `ritardaDiUnGiorno`) incrementa la data di un oggetto `ProgrammazioneFilm` di un giorno, con un codice più compatto (e meno leggibile). Poi abbiamo istanziato un oggetto di tipo `Film` che abbiamo chiamato `shining`. Subito dopo abbiamo creato un oggetto di tipo `ProgrammazioneFilm` che abbiamo chiamato `programmazioneShining`, che definisce per `shining` un orario e una data. Poi abbiamo ottenuto un oggetto `UnaryOperator` composto dalle due espressioni lambda che abbiamo creato precedentemente, e l'abbiamo chiamato significativamente `ritardaDiUnOraEUnGiorno`. Infine abbiamo stampato un oggetto di tipo `ProgrammazioneFilm`, risultante dall'applicazione dell'`UnaryOperator` composto `ritardaDiUnOraEUnGiorno`, sull'oggetto `shining`. L'output è il seguente:

```
Shining verrà proiettato il 2018-05-30 alle 22:00
```

Il vantaggio di aver concatenato in un unico `UnaryOperator` le due modifiche all'oggetto `ProgrammazioneFilm`, ci ha permesso di non fare passaggi intermedi e di ottenere direttamente lo scopo desiderato.

Riepilogo

Java supporta anche il paradigma funzionale grazie alle **espressioni lambda** e ai **reference a metodi.** La sintassi delle espressioni lambda è progettata per scrivere meno codice possibile, e lo stesso può dirsi per i reference a metodi. Abbiamo visto come questi due concetti siano intercambiabili e come il programmatore abbia la scelta di usare queste due sintassi a seconda di quella che può risultare più comoda al momento. Le espressioni lambda (e quindi i reference a metodi) possono essere utilizzate in luogo delle classi anonime nei casi in cui ci sia un unico metodo astratto da ridefinire e che la sua implementazione non sia costituita da troppe righe di codice. Un**'espressione lambda** sostituisce l'implementazione di un'interfaccia funzionale, ridefinendo il codice del suo metodo SAM. Le espressioni lambda essenzialmente forniscono, oltre alla possibilità di passare codice ad un metodo (cosa che si poteva fare già con una classe anonima), anche la sinteticità e la semplicità (visto che non bisogna tener conto di tutte le regole che esistono per le classi anonime). Inoltre è importante capire che quando passiamo una espressione lambda ad un metodo, permettiamo a quel metodo di definire quando invocare il codice passato. Dovremmo usare le espressioni lambda quando il nostro obiettivo è quello di passare in maniera dinamica un certo algoritmo ad un altro metodo.

Un **reference a metodo** può essere passato come parametro ad un metodo, esattamente come una espressione lambda. È molto utile utilizzare un reference a metodo in luogo di una espressione lambda quando questo deve essere riutilizzato, per evitare di scrivere più volte le stesse righe di codice. Inoltre se ci sono troppe righe di codice da definire, è preferibile utilizzare un reference a metodo piuttosto che un'espressione lambda. Esistono quattro tipi di sintassi diverse per utilizzare i reference a metodo. La sintassi del **reference a un metodo statico** permette di passare un metodo definito statico come parametro ad un metodo. In questo caso si utilizza il nome del tipo per referenziare il metodo, seguito come sempre dalla coppia `::` e dal nome del metodo statico: `nomeTipo::nomeMetodoStatico`. Stesso discorso per un **reference a un metodo d'istanza**, dove però è necessario istanziare un oggetto ed utilizzare il suo reference prima della coppia `::`, seguito dal nome del metodo d'istanza: `nomeOggetto::nomeMetodoDIstanza`. Si può utilizzare la sintassi di un **reference a un metodo d'istanza di un**

certo tipo, quando il compilatore può dedurre senza l'istanza di un particolare oggetto, quali sono gli input e gli output del metodo. In questo caso si utilizza il nome del tipo, la coppia `::` e il nome di un metodo non statico: `nomeTipo::nomeMetodoDIstanza`. Infine è possibile anche utilizzare la sintassi per il **reference ad un costruttore**, che consente di passare una nuova istanza come parametro ad un metodo. La sintassi utilizza il nome del tipo, la coppia `::` e infine la parola chiave `new`: `nomeTipo::new`.

Abbiamo introdotto le interfacce funzionali principali del package `java.util.function`: `Predicate`, `Consumer`, `Supplier`, `Function` e `UnaryOperator`. L'interfaccia funzionale **`Predicate`**, dichiarando il metodo `boolean test(T t);` ben si adatta alle situazioni in cui bisogna fare un test che ritorna un booleano. L'interfaccia funzionale **`Consumer`**, dichiarando il metodo `void accept(T t);` che non ha tipo di ritorno, si adatta bene alle funzionalità di aggiornamento dello stato interno dell'oggetto che gli viene passato, o comunque nei casi in cui lo si vuole utilizzare magari chiamando i suoi metodi. L'interfaccia funzionale **`Supplier`**, dichiarando il metodo `T get();` si adatta bene nei contesti in cui serve gestire un metodo factory. Il suo metodo `get()` restituisce un istanza del tipo parametro dichiarato. L'interfaccia funzionale **`Function`**, dichiarando il metodo `R apply(T t);` astrae proprio il concetto di funzione più classico, dove c'è un input (impersonificato dal tipo parametro `T`) e un output (tipo parametro `R`). Infine l'interfaccia funzionale **`UnaryOperator`**, non è altro che un'estensione di `Function`, che dichiara il metodo `apply()` dove il tipo di ritorno è uguale al parametro `T apply(T t);`. Quest'interfaccia astrae un'operazione di tipo *trasformazione*, che può essere utile quando si dovranno combinare più espressioni lambda per eseguirle tutte insieme.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere la sintassi e le modalità d’uso delle espressioni lambda (unità 16.1)	☐	
Saper decidere quando utilizzare un’espressione lambda o una classe anonima (unità 16.1)	☐	
Saper gestire la dinamicità del passaggio di codice ad un metodo con una espressione lambda (unità 16.1)	☐	
Saper definire ed utilizzare tutte le quattro diverse sintassi dei reference a metodi (unità 16.2)	☐	
Saper utilizzare le interfacce funzionali definite nel package `java.util.function` (unità 16.3)	☐	

Note:

17

Collections Framework e Stream API

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Comprendere l'utilità del framework Collections (unità 17.1).
- ✔ Conoscere le principali interfacce del framework Collections e i metodi che espongono (unità 17.2, 17.3, 17.4, 17.5, 17.6).
- ✔ Conoscere le principali implementazioni del framework Collections e i metodi che espongono (unità 17.2, 17.3, 17.4, 17.5, 17.6).
- ✔ Saper utilizzare le classi d'utilità `Arrays` e `Collections` e i metodi che espongono (unità 17.7).
- ✔ Comprendere l'utilità della libreria Stream API (unità 17.8).
- ✔ Saper utilizzare stream, classi `Optional` e pipeline (unità 17.8).
- ✔ Saper utilizzare metodi di aggregazione, di riduzione e comprendere gli stream paralleli (unità 17.8).

In questo capitolo introdurremo finalmente uno degli argomenti più importanti per il programmatore Java: il framework **Collections.** Alcune classi del framework sono state presentate già dai primi capitoli. L'argomento

viene approfondito solo ora per ragioni di metodologia didattica, non certo perché poco rilevante. È quasi impossibile infatti trovare un programma Java (almeno che non si tratti di un programma particolarmente banale) che non faccia uso di questo framework. Anche nei semplici esempi che abbiamo presentato sino ad ora siamo stati costretti ad anticipare alcune delle fondamentali implementazioni di questo framework, come le interfacce `List` e `Map`, e le loro implementazioni `ArrayList` ed `HashMap`. Il framework Collections è in buona parte contenuto nel package `java.util`, ma si estende anche nel package `java.util.concurrent`, con una serie di interfacce e classi che definiscono collezioni per gestire la concorrenza dei processi.

Con il termine "framework" in questo caso intendiamo un insieme di classi e di interfacce riutilizzabili ed estendibili. Questa definizione è un po' semplicistica ma per i nostri scopi può andare bene.

In questo capitolo introdurremo anche la libreria denominata **Stream API.** Questa libreria, introdotta nella versione 8 di Java, è essenzialmente basata sull'interfaccia `Stream`, definita nel package `java.util.stream`, e ci permetterà di utilizzare le collezioni in maniera più semplice ed efficiente. Gli stream sono stati progettati per essere utilizzati congiuntamente alle espressioni lambda e ai riferimenti a metodi, quindi è fondamentale che il lettore abbia già studiato il capitolo su questi argomenti.

Il framework Collections è stato introdotto in Java nella versione 1.2, portando il linguaggio ad un livello molto più alto rispetto a prima. Ma già nella versione 1.0 esistevano delle collezioni, tanto che queste sono in qualche modo considerate facenti parte del framework. Stiamo parlando in particolare delle classi `Hashtable` **e** `Vector`, **e dell'interfaccia** `Enumeration`. **Queste sono spiegate nell'appendice on line O e sono citate all'interno di questo capitolo, per fare alcuni confronti.**

17.1 Introduzione al Framework Collections

Il framework noto come Collections è costituito da una serie di classi, interfacce e algoritmi per gestire collezioni. Una **collezione** è definibile come un oggetto che raggruppa più elementi in una singola entità. I vantaggi di avere a disposizione questo framework per la programmazione sono tanti: possibilità di scrivere meno codice, incremento delle prestazioni, interoperabilità tra classi non relazionate tra loro, riusabilità, algoritmi complessi già implementati a disposizione, maggiore qualità e semplicità del codice, estensibilità per creare nuove librerie.

Il framework è basato su otto interfacce principali che vengono poi estese da tante altre classi astratte e concrete. In figura 17.1 viene illustrata, tramite un diagramma delle classi UML semplificato (cfr. appendice I per definizione UML e appendice L per sintassi UML), la gerarchia di queste interfacce che è costituita da due alberi diversi.

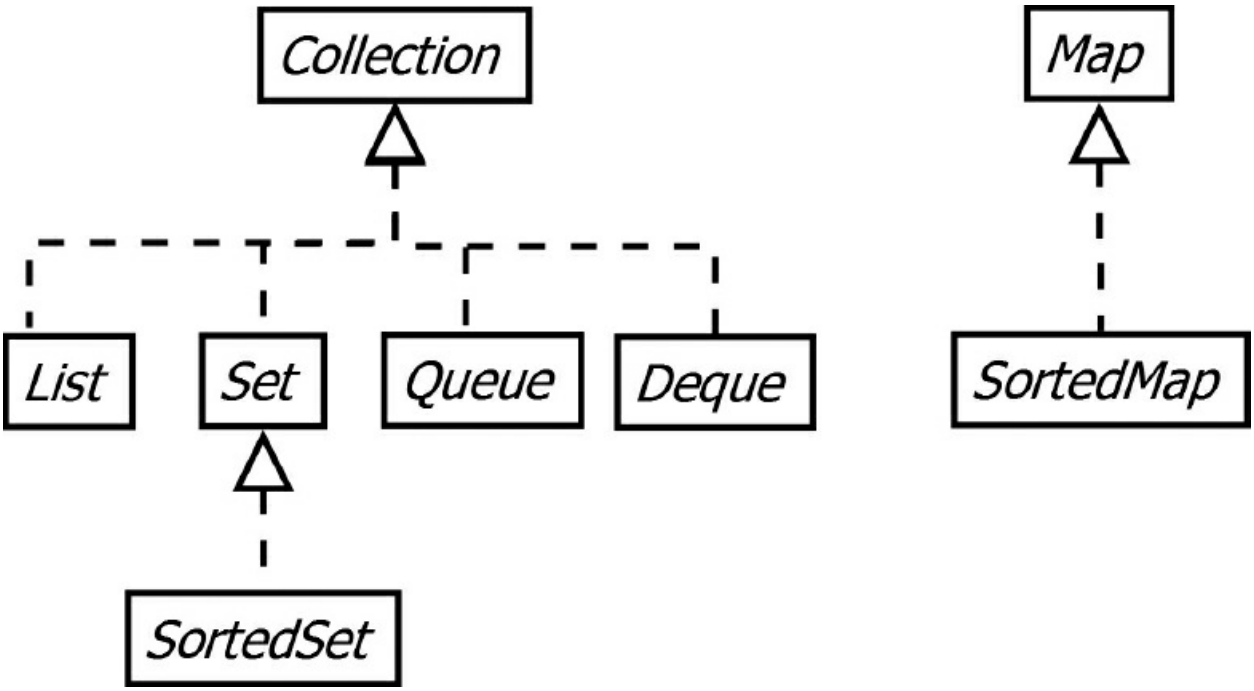


Figura 17.1 - Le gerarchie tra le interfacce fondamentali del framework Collections.

Ad un primo livello si trovano le interfacce `Collection` (la più importante) e `Map`. `Collection` è estesa dalle interfacce `List`, `Queue`, `Deque` e `Set`. Quest'ultima a sua volta è estesa da `SortedSet`. `Map` è invece estesa da `SortedMap`. Ognuna di queste interfacce possiede proprietà che le sottoclassi

ereditano. Per esempio, una classe che implementa `Set` (in italiano “insieme”) erediterà il fatto di essere un tipo di collezione non ordinata che non accetta elementi duplicati. Cerchiamo di fare chiarezza con una breve panoramica su queste interfacce e le loro proprietà.
L’interfaccia `Collection` è la radice principale di tutta la gerarchia del framework. Essa astrae il concetto di *insiemi di oggetti*, detti *elementi*. Esistono implementazioni che ammettono elementi duplicati ed altre che non lo permettono, collezioni ordinate e non ordinate. La libreria non mette a disposizione alcuna implementazione diretta di `Collection`, ma solo delle sue dirette sottointerfacce come `Set` e `List`. L’interfaccia `Collection` viene definita come il minimo comune denominatore che tutte le collection devono implementare. Un `Set` è un tipo di collection che, astraendo il concetto di **insieme matematico**, non ammette elementi duplicati. Una `List` è una collezione ordinata (a volte viene detta anche **sequence).** In una lista viene sempre associato un indice ad ogni elemento, che equivale alla posizione dell’elemento stesso all’interno della lista. Una lista ammette elementi duplicati (distinguibili fra loro per la posizione).

L’interfaccia `Queue`, fornisce invece operazioni di inserimento, estrazione e ricerca, e gli elementi sono organizzati come in una **pila FIFO** (acronimo di **First In First Out)**, ovvero una collezione dove il primo elemento che viene inserito, viene considerato il primo elemento della collezione.

L’interfaccia `Deque` (che è l’abbreviativo di **double ended queue** e si pronuncia “deck”) fornisce le stesse operazioni dell’interfaccia `Queue` ma utilizzabili ad entrambe le estremità della pila, ed i suoi elementi possono essere organizzati sia con la filosofia FIFO sia con la filosofia **LIFO** (acronimo di **Last In First Out)**, ovvero dove l’ultimo elemento inserito viene considerato il primo elemento della collezione. Una `Map` è una collezione che associa chiavi ai suoi elementi. Le mappe non possono contenere chiavi duplicate, ed ogni chiave può essere associata ad un solo valore. Le ultime due interfacce, `SortedSet` e `SortedMap`, rappresentano le versioni ordinate di `Set` e `Map`. Aggiungono diverse nuove funzionalità relative all’ordinamento, che avviene sempre tramite l’implementazione da parte degli elementi dell’interfaccia `Comparable`, oppure tramite la codifica di oggetti `Comparator` (cfr. paragrafo 12.8.1). Tali interfacce possono definire metodi implementati o statici, e quindi troveremo metodi ereditati da queste

interfacce già pronti per l'uso.

> **Praticamente tutte le classi e le interfacce di questo framework sono generiche, e se non usate specificando i tipi parametri, saranno considerate "raw type" dal compilatore, che quindi ci segnalerà dei warning. Bisognerebbe quindi specificare sempre il tipo parametro per le collections. Possiamo affermare che i tipi generici sono usati soprattutto per le collections.**

Nei prossimi paragrafi andremo quindi ad esplorare queste interfacce e le loro più importanti implementazioni. L'introduzione della libreria Stream API ha rivoluzionato completamente il modo in cui usiamo le collection. Quindi introdurremo tale libreria nella maniera più pratica possibile.

17.2 L'interfaccia `Collection`

È la superinterfaccia per definizione del framework. Possiamo usare un reference di `Collection` per puntare a tutte le implementazioni che appartengono allo stesso albero gerarchico per il polimorfismo per metodi. Per esempio possiamo scrivere:

```
Collection<String> collection = new ArrayList<>();
```

Questa pratica che sfrutta il polimorfismo per dati è sempre consigliabile, e ancora di più per le collections. Per esempio se avessimo scritto tante altre righe di un programma che utilizzano la variabile `collection`, e avessimo necessità di voler cambiare il tipo dell'oggetto per esempio con una `LinkedList`, l'unica riga da cambiare sarebbe proprio quella appena scritta. Tutte le altre parti di codice che usano collection continueranno a funzionare. Essendo `ArrayList` e `LinkedList` implementazioni di `List` (che a sua volta estende `Collection`), a seconda dei casi, è possibile che sia più utile

utilizzare il polimorfismo per dati usando tale interfaccia piuttosto che `Collection`. Per esempio se vogliamo sfruttare il metodo `get()` definito nell'interfaccia `List`, che prende in input un intero rappresentante l'indice dell'elemento della lista che si vuole recuperare, allora è preferibile usare un reference di `List` piuttosto che di `Collection`:

```
List<String> list = new ArrayList<>();
list.add("Primo elemento");
list.add("Secondo elemento");
String primoElemento = list.get(0);
```

Infatti un reference di tipo `Collection` non può invocare il metodo `get()`, e dovremmo per forza ricorrere ad un cast.

17.2.1 Metodi di `Collection`

L'interfaccia `Collection` definisce metodi di base come:

- `int size()` ritorna il numero degli elementi contenuti nella collezione.
- `boolean add(E element)` aggiunge un elemento (del tipo parametro) alla collezione, e restituisce `true` o `false` a seconda del fatto che l'operazione sia andata a buon fine o meno.
- `boolean remove(Object element)` rimuove un elemento dalla collezione, e restituisce `true` o `false` a seconda del fatto che l'operazione sia andata a buon fine o meno.
- `boolean isEmpty()` restituisce `true` o `false` a seconda del fatto che la collection contenga elementi o meno.
- `void clear()` elimina tutti gli elementi dalla collezione.
- `boolean contains(Object element)` restituisce `true` o `false` a seconda del fatto che l'elemento specificato in input sia contenuto nella collection o meno. Per verificare l'uguaglianza viene utilizzato il metodo `equals()`.
- `Iterator<E> iterator()` restituisce un'implementazione di un'interfaccia `Iterator` che serve per iterare sugli elementi della collezione (cfr. prossimi paragrafi). Per esempio il seguente

frammento di codice:

```
Collection<String> progBands = new HashSet<>();
progBands.add("Dream Theater");
progBands.add("Ayreon");
progBands.add("Pain Of Salvation");
Iterator<String> iterator = progBands.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

stamperà:

```
Ayreon
Dream Theater
Pain Of Salvation
```

Si noti che `HashSet` (come tutti gli oggetti `Set` che non implementano `SortedSet`) non mantiene l'ordine in cui sono stati inseriti gli elementi.

Inoltre `Collection` definisce metodi per interagire con altre collezioni (il che significa con implementazioni di `List`, `Set` e così via):

- `boolean containsAll(Collection<?> c)` restituisce `true` o `false` a seconda del fatto che gli elementi della collezione specificata in input siano tutti contenuti nella collection su cui viene chiamato il metodo. Questo indipendentemente dall'ordine degli oggetti nelle collezioni. Per esempio il seguente frammento di codice:

```
Collection<Integer> collection1 = new ArrayList<>();
for (int i = 0; i < 5; i++) {
    collection1.add(i);
}
Collection<Integer> collection2 = new ArrayList<>();
collection2.add(2);
collection2.add(1);
collection2.add(4);
System.out.println(collection1.containsAll(collection2));
```

stamperà:

```
true
```

- `boolean addAll(Collection<? extends E> c)` aggiunge tutti gli

elementi della collezione specificata in input alla collezione su cui è stato chiamato il metodo, e restituisce `true` o `false` a seconda del fatto che l'operazione sia andata a buon fine o meno.

- `boolean removeAll(Collection<?> c)` rimuove tutti gli elementi della collezione specificata in input alla collezione su cui è stato chiamato il metodo, e restituisce `true` o `false` a seconda del fatto che l'operazione sia andata a buon fine o meno.
- `boolean retainAll(Collection<?> c)` conserva tutti gli elementi della collezione specificata in input nella collezione su cui è stato chiamato il metodo (rimuovendo gli altri), e restituisce `true` o `false` a seconda del fatto che l'operazione sia andata a buon fine o meno. Per esempio, considerando le due collezioni inizializzate come nell'esercizio precedente, allora il seguente frammento di codice:

```
collection1.retainAll(collection2);
System.out.println(collection1);
```

stamperà:

```
[1, 2, 4]
```

> **Per questa stampa è stato invocato automaticamente il metodo `toString()` di `ArrayList`, che ha formattato gli elementi della collection in maniera leggibile, separandoli con virgole e includendoli tra parentesi quadre.**

L'interfaccia `Collection` definisce anche metodi per interagire con array:

- `Object[] toArray()` che restituisce un array di `Object` a partire da una collezione.
- `<T> T[] toArray(T[] a)` che restituisce un array di oggetti di tipo parametro a partire da una collezione. Per esempio considerando l'oggetto `progBands` dell'esempio sul metodo `iterator()`, con il seguente codice andiamo a riempire un array di stringhe con il

contenuto della collection, e il successivo ciclo foreach produrrà lo stesso output del ciclo che abbiamo visto prima con l'iterator:

```
String [] progBandsArray = progBands.toArray(new String[0]);
for (String progBand :  progBandsArray) {
    System.out.println(progBand);
}
```

> **È possibile anche il processo inverso per riempire una collection (in particolare una lista) a partire da un array (cfr. metodo `asList()` della classe di utilità `Arrays`).**

Infine, la novità importante che ha portato Java 8, è che questa interfaccia espone i metodi `stream()` e `parallelStream()`. Questi restituiscono oggetti di tipo `Stream` che consentono di ciclare in maniera sequenziale o parallela sulla collezione. Gli stream si fanno preferire rispetto agli iteratori e ai costrutti foreach, per le loro prestazioni e il loro supporto alle espressioni lambda. Quindi gli stream permettono di scrivere meno codice con più efficienza. Approfondiremo il discorso sugli stream nella seconda parte di questo capitolo.

17.2.2 Iterare sulle collezioni

Esistono tre modi per iterare sulle collezioni:

1. usando cicli foreach;
2. usando iteratori;
3. usando il metodo di default `forEach()` definito nell'interfaccia `Collection` e sfruttando le cosiddette **operazioni di aggregazione** (in inglese **aggregate operations)** della nuova libreria denominata Stream API.

Questo vale per tutte le collezioni, ma implementazioni più specifiche possono usare anche tipologie di iterazioni come cicli `for` ordinari sfruttando un indice (è il caso delle liste).

17.2.2.1 Ciclo foreach

L'interfaccia `Collection` estende l'interfaccia `Iterable`, e tutto quello che estende `Iterable` può essere iterato tramite un ciclo foreach (nell'appendice on line O è spiegato anche come implementare `Iterable`). Per esempio considerando sempre la collection `progBands` già usata nei precedenti esempi, possiamo iterare su di essa anche in questo modo:

```
for (String progBand :  progBands) {
    System.out.println(progBand);
}
```

Bisognerebbe usare questo tipo di costrutto quando abbiamo bisogno di iterare su collezioni di piccole dimensioni e non si devono fare operazioni impegnative sugli elementi.

17.2.2.2 Iteratori

Gli **iteratori** sono oggetti cha astraggono dei cursori per eseguire cicli sugli elementi di una collezione. Essi possono spostarsi elemento per elemento (uno alla volta) partendo dal primo e arrivando all'ultimo, e in generale non possono cambiare direzione e tornare indietro. L'interfaccia `Iterator` come abbiamo già visto, viene usata sfruttando il metodo `iterator()` di `Collection`. Infatti ogni implementazione ha la sua istanza di `Iterator`. Per esempio, sempre considerando l'oggetto `progBands` degli esempi precedenti, il seguente codice:

```
Iterator<String> iterator = progBands.iterator();
System.out.println(iterator.getName());
```

stamperà:

```
java.util.HashMap$KeyIterator
```

Ovvero, tramite il metodo `getName()` chiamato sull'oggetto `Class` che ci ha ritornato il metodo `getClass()`, abbiamo scoperto che stiamo usando un oggetto istanziato da una classe innestata di `HashMap` che si chiama `KeyIterator` (ricordiamo che il simbolo `$` viene utilizzato per le classi interne, cfr. paragrafo 10.1.2). Fortunatamente grazie all'astrazione fornita

dal polimorfismo per dati, non ci interessa conoscere questa implementazione. Semplicemente ci limiteremo sempre ad usare i metodi che appartengono all'interfaccia `Iterator`.

> **Per cultura e completezza è utile sapere che, nonostante possa sembrare strano al neofita adoperare un iteratore per eseguire un ciclo su una collezione (il ciclo foreach è sicuramente più intuitivo e semplice da utilizzare), "Iterator" è anche il nome di un famoso Design Pattern sul cui modello è stata creata quest'interfaccia. Come tanti altri pattern questa soluzione si basa su uno dei principi fondamentali dell'Object Orientation: il "Dependency Invertion Principle" (conosciuto anche con il suo acronimo "DIP") che in italiano si traduce come "Principio di inversione della dipendenza". Questo principio ben formulato da Robert C. Martin ci consiglia di far dipendere le nostre classi dalle astrazioni e non dai dettagli concreti delle implementazioni. Infatti nell'esempio dell'`Iterator` utilizziamo una classe innestata `KeyIterator` di `HashMap`, ignorandone l'esistenza. Il DIP è essenzialmente il principio che ispira il polimorfismo stesso.**

L'interfaccia `Iterator` definisce solo tre metodi molto semplici. Segue la sua dichiarazione epurata di tutto quello che non ci serve per ora:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Il metodo `hasNext()` l'abbiamo già usato nell'esempio che abbiamo fatto quando abbiamo presentato il metodo `iterator()` di `Collection`. Esso restituisce `true` se l'iteratore non si trova sull'ultimo elemento della collezione, altrimenti restituisce `false` (perché dopo non ci sono più elementi). Il metodo `next()` lo abbiamo utilizzato per far avanzare di un

passo l'iteratore. Infine il metodo `remove()` rimuove l'elemento corrente (ovvero dove si trova l'iteratore). È molto importante sapere che questo metodo è l'unico che permette la rimozione di un elemento da una collezione durante un'iterazione in maniera thread-safe. Ogni altro tentativo non è sicuro. Ecco un esempio banale dove con un iteratore rimuoviamo un elemento della collezione `progBands`:

```
Iterator<String> iterator = progBands.iterator();
while (iterator.hasNext()) {
    if (iterator.next().equals("Ayreon")) {
        iterator.remove();
    }
}
System.out.println(progBands);
```

Il cui output sarà:

```
[Dream Theater, Pain Of Salvation]
```

Esiste anche un quarto metodo introdotto con Java 8. Il metodo `forEachRemaining()` prende in input un oggetto di tipo `Consumer` (cfr. paragrafo 16.3.2). È un metodo di default, quindi ha già un'implementazione, ma questa ci interessa relativamente. Esso esegue l'operazione specificata per tutti gli elementi su cui bisogna ancora eseguire il ciclo, fino a quando sono finiti oppure se scatta un'eccezione. Per esempio, consideriamo la seguente classe `Smartphone`:

```
public class Smartphone {
    private String modello;
    private String marca;
    private int prezzo;
    // Costruttori e metodi set e get omessi
    public String toString() {
        return marca + " " + modello;
   }
}
```

Il seguente codice:

```
Collection<Smartphone> smartphones = new HashSet<>();
smartphones.add(new Smartphone("Samsung", "Note 74"));
smartphones.add(new Smartphone("Apple", "IPhone 225"));
```

```
smartphones.add(new Smartphone("Motorola", "G55"));
Iterator<Smartphone> iterator = smartphones.iterator();
iterator.forEachRemaining(element ->
{element.setMarca("Google");});
System.out.println(smartphones);
```

stamperà:

```
[Google G55, Google Note 74, Google IPhone 225]
```

Si noti che abbiamo modificato la marca di tutti gli smartphone con un'unica chiamata al metodo `forEachRemaining()` ed usando una banale espressione lambda. In generale bisognerebbe usare un `Iterator` in luogo di un ciclo foreach quando c'è bisogno di rimuovere elementi dalla collection (il ciclo foreach non mette a disposizione un `Iterator`, e di conseguenza non c'è metodo `remove()` da invocare).

17.2.2.3 Metodo `forEach()` dell'interfaccia `Iterable`

Con l'avvento di Java 8, è stato introdotto il metodo `forEach()` come metodo di default nell'interfaccia `Iterable` (da non confondere con il costrutto che di solito chiamiamo foreach). Il metodo `forEach()` prende in input un oggetto di tipo `Consumer` (cfr. paragrafo 16.3.2) che ricordiamo, ci permette di eseguire operazioni su oggetti (di solito per modificarne lo stato interno). Questo significa che qualsiasi collezione potrà iterare direttamente sui suoi elementi semplicemente chiamando questo metodo, e passandogli in input un'espressione lambda per fare operazioni sugli stessi elementi. Per esempio, considerando l'oggetto `smartphones` dell'esempio precedente, con il seguente statement:

```
smartphones.forEach(s->System.out.println(s));
```

saranno stampati tutti gli elementi della collezione. Infatti con l'espressione lambda specificata, abbiamo fatto in modo che ogni elemento fosse stampato, perché il metodo `forEach()` eseguirà l'espressione lambda per ogni elemento della collezione.

17.3 Interfaccia `List`

L'interfaccia `List` rappresenta una collezione ordinata e indicizzata di elementi in cui sono ammessi duplicati. Estende `Collection`, che a sua volta estende `Iterable`, e naturalmente eredita tutti i metodi pubblici da entrambe. In generale una lista definisce i metodi `add()` e `addAll()` (che ricordiamo prende in input una collection), in modo tale che aggiungano in coda gli elementi in input. La sequenza di come gli elementi di una lista vengono aggiunti quindi, definisce l'ordine della lista. Uno dei metodi più utilizzati in assoluto è il metodo `get()` che prende in input un indice intero e restituisce l'oggetto corrispondente a quell'indice. Si usa per esempio quando si cicla su una lista con un ciclo `for` sfruttando l'indice della collection:

```
List<Integer> list = new ArrayList<>(3);
list.add(25);
list.add(7);
list.add(74);
int size = list.size();
for (int i = 0; i < size; i++) {
    System.out.println(list.get(i));
}
```

Esiste anche il metodo `add()` che prende in input non solo l'elemento da aggiungere ma anche l'indice della posizione dove deve essere inserito, facendo spostare in avanti tutti gli altri elementi se ve ne sono. Equivalentemente è definito il metodo `set()` che prende in input sia l'elemento da impostare che l'indice della posizione dove deve essere impostato, ma questa volta l'elemento in input va semplicemente a sovrascrivere l'eventuale elemento già esistente a quell'indice. Interessante anche la possibilità di ottenere mediante la chiamata al metodo `listIterator()`, un oggetto di tipo `ListIterator` che a differenza di un `Iterator` classico, consente di iterare in entrambe le direzioni. Infatti definisce metodi come `hasPrevious()` e `previous()` che sono gli equivalenti di `hasNext()` e `next()` ma si riferiscono all'elemento precedente invece che al successivo.

17.3.1 Implementazioni di `List`

Le principali implementazioni di `List` sono `ArrayList`, e `LinkedList`. `ArrayList` è in assoluto la collection più utilizzata, ed è stata creata proprio

per rappresentare una evoluzione ridimensionabile di un array, oltre che per sostituire la superata classe `Vector` (cfr. appendice on line O). `Vector` e `ArrayList` hanno le stesse funzionalità, ma quest'ultima è più performante perché non è sincronizzata. `ArrayList` ha prestazioni nettamente superiori anche a `LinkedList`, che conviene utilizzare solo per gestire collezioni di tipo *coda*. Si tratta di una cosiddetta **lista concatenata**, nella quale ogni elemento mantiene un reference verso l'elemento successivo e l'elemento precedente. Mette a disposizione tra l'altro metodi come `addFirst()`, `getFirst()`, `removeFirst()`, `addLast()`, `getLast()` e `removeLast()`. È quindi opportuno scegliere l'utilizzo di una `LinkedList` in luogo di `ArrayList` solo quando si devono aggiungere spesso elementi all'inizio della lista, oppure quando si vogliono eliminare elementi all'interno della lista durante le iterazioni. Queste operazioni richiedono infatti un **tempo costante** nelle `LinkedList` e un **tempo lineare** (ovvero che dipende dal numero degli elementi) in un `ArrayList`. In compenso però, l'accesso posizionale in una `LinkedList` è lineare, mentre è costante in un `ArrayList`. Questo implica una prestazione superiore da parte di `ArrayList` nella maggior parte dei casi.

> **La classe `LinkedList` implementa anche l'interfaccia `Queue`, una delle interfacce base (introdotta nella versione 1.5), e l'interfaccia `Deque` (introdotta dalla versione 1.6). `Deque` è un termine abbreviativo per definire una "double ended queue". I metodi `addFirst()`, `getFirst()`, `removeFirst()`, `addLast()`, `getLast()` e `removeLast()` di cui sopra non sono altro che implementazioni dei metodi astratti di `Deque`.**

Anche l'`ArrayList` possiede un parametro di configurazione: la capacità iniziale. Se istanziamo un `ArrayList` con capacità iniziale `20`, avremo un oggetto con `20` posizioni vuote (ovvero in ogni posizione c'è un reference che punta a `null`) pronte a essere riempite. Quando sarà aggiunto il ventunesimo elemento, l'`ArrayList` si ridimensionerà automaticamente per avere capacità ventuno, e questo avverrà per ogni nuovo elemento. Per ogni nuovo elemento che si vuole aggiungere oltre la capacità iniziale, l'`ArrayList` dovrà prima

ridimensionarsi e poi aggiungere l'elemento. Questa doppia operazione porterà ad un decadimento delle prestazioni.

È però possibile ottimizzare le prestazioni di una `ArrayList` nel caso si vogliano aggiungere nuovi elementi superata la capacità iniziale. Infatti, quest'ultima si può modificare a piacimento al volo, in modo tale che l'`ArrayList` non sia costretto a ridimensionarsi per ogni nuovo elemento. Per fare ciò, basta utilizzare il metodo `ensureCapacity()` passandogli la nuova capacità, prima di chiamare il metodo `add()`. Per avere un'idea di quanto sia importante ottimizzare un `ArrayList`, viene presentato un semplice esempio. Sfruttiamo il metodo statico `currentTimeMillis()` della classe `System` (cfr. paragrafo 12.2.3) per calcolare i millisecondi che impiega un ciclo a riempire l'`ArrayList`:

```
  //capacità iniziale 1
  ArrayList<String> list = new ArrayList<>(1);
  long startTime = System.currentTimeMillis();
  list.ensureCapacity(100000000);
  for (int i = 0; i < 100000000; i++) {
      list.add("nuovo elemento");
  }
  long endTime = System.currentTimeMillis();
  System.out.println("Tempo = " + (endTime - startTime));
```

L'output è:

```
Tempo = 347
```

Commentando la riga:

```
  list.ensureCapacity(100000000);
```

l'output cambierà:

```
Tempo = 1208
```

La prestazione è nettamente peggiore e la differenza sarà ancora più evidente aumentando il numero di elementi.

Se rimuoviamo un elemento da un `ArrayList`, la sua capacità non diminuisce. Esiste il metodo `trimToSize()` per ridurre la capacità dell'`ArrayList` al numero degli elementi effettivi.

17.4 Le interfacce `Set` e `SortedSet`

L'interfaccia `Set`, (in italiano "insieme" nel senso di **insieme matematico)** rappresenta una collezione non ordinata di elementi in cui non sono ammessi duplicati. In particolare l'interfaccia `Set` non definisce altri metodi rispetto a `Collection`, bensì impone regole più restrittive quando questi vengono utilizzati. Questo significa che se per esempio aggiungiamo due volte tramite il metodo `add()` lo stesso elemento, questo verrà aggiunto solamente una volta.

L'interfaccia `SortedSet`, invece rappresenta una collezione ordinata di elementi in cui non sono ammessi duplicati. Questa definisce alcuni metodi per lavorare su range di elementi o sugli estremi della collezione.

17.4.1 Implementazioni di `Set` e `SortedSet`

Un'implementazione di `Set` è `HashSet`, mentre un'implementazione di `SortedSet` è `TreeSet`. `HashSet` è più performante rispetto a `TreeSet` ma non gestisce l'ordinamento. Entrambe queste classi non ammettono elementi duplicati e sono di sicuro le più utilizzate implementazioni di insiemi. `HashSet` garantisce prestazioni migliori e dovrebbe essere sempre preferita a `TreeSet`, a meno che non si voglia iterare in maniera ordinata sugli elementi della collezione. Segue un esempio di utilizzo di `TreeSet` dove si aggiungono elementi di tipo stringa, per poi stamparli sfruttando il metodo `forEach()` con un reference al metodo `println()` di `System.out`:

```
Set<String> set = new TreeSet<>();
set.add("c");
set.add("a");
set.add("b");
set.add("b");
set.forEach(System.out::println);
```

L'output sarà:

```
a
b
c
```

Infatti l'elemento duplicato (`b`) non è stato aggiunto e gli elementi sono stati

ordinati secondo la loro natura di stringhe. Per ordinare la collection è stato utilizzato il metodo `compareTo()` dell'interfaccia `Comparable` implementata da `String`. Ma esiste anche un costruttore di `TreeSet` che prende in input un `Comparator`, per far sì che la collezione sia ordinata secondo un metodo personalizzato. Se istanzias-simo l'oggetto `TreeSet` nel seguente modo passandogli un'espressione lambda che inverte l'algoritmo di ordinamento naturale:

```
TreeSet<String> set = new TreeSet<>((x,y) -> {return -
(x.compareTo(y));});
```

otterremmo l'output:

```
c
b
a
```

Dalla versione 1.6 gli oggetti `TreeSet` sono anche *bidirezionali*. Infatti, è possibile anche ottenere un'istanza di `Iterator` che itera gli elementi del `TreeSet` dall'ultimo al primo invocando il metodo `descendingIterator()`:

```
Iterator iter = set.descendingIterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

Se avessimo istanziato un `HashSet` in luogo di `TreeSet`:

```
Set<String> set = new HashSet<>();
```

avremmo ottenuto un ordinamento non prevedibile.

Un'altra implementazione degna di nota è `LinkedHashSet` introdotta nella versione 1.4. È una collezione che può essere considerata un po' un `Hashtable` (di cui parliamo nell'appendice O) e un po' una `LinkedList`. Rispetto ad un `HashSet` questa mantiene l'ordine degli elementi compatibile con il loro ordine di inserimento, come fanno le liste. Se nell'esempio precedente avessimo istanziato un `LinkedHashSet` in luogo di un `TreeSet`:

```
Set<String> set = new LinkedHashSet<>();
```

avremmo ottenuto il seguente output:

```
c
a
b
```

È interessante notare nella versione 6 di Java sia stata introdotta una nuova interfaccia che estende `SortedSet`: `NavigableSet`. Il discorso è molto simile a quello che faremo per `NavigableMap`. Quest'interfaccia definisce nuovi metodi per una navigazione della collezione come `lower()`, `floor()`, `ceiling()` e `higher()`, che restituiscono rispettivamente un elemento minore, minore o uguale, maggiore o uguale e maggiore dell'elemento specificato come argomento. La classe `TreeSet` implementa proprio quest'interfaccia e il metodo `descendingIterator()` è anch'esso definito nell'interfaccia `NavigableSet`.

17.5 Le interfacce `Queue` e `Deque`

L'interfaccia `Queue` (in italiano "coda") rappresenta una collezione progettata per raccogliere elementi prima di essere processata. Estende l'interfaccia `Collection` definendo nuovi metodi per l'inserimento, la rimozione e l'utilizzo dei dati. Ognuno di questi metodi è presente in due formati differenti: se l'operazione fallisce, un formato lancia un'eccezione, mentre l'altro restituisce un valore speciale (per esempio `null` o `false`). In particolare quando parliamo di valore speciale ci riferiamo alla situazione in cui il metodo restituisce o l'oggetto stesso appena aggiunto o recuperato, oppure un booleano (come nel caso del metodo `offer()`). Quindi, a seconda dell'esigenza, lo sviluppatore può usufruire di un metodo piuttosto che di un altro. La seguente tabella riassume quanto appena asserito:

Tipo di operazione	Lancia eccezione	Ritorna valore speciale
Inserimento	`add(e)`	`offer(e)`
Rimozione	`remove()`	`poll()`
Recupero	`element()`	`peek()`

Le collection sono tutte ridimensionabili e quindi in generale l'inserimento è

sempre possibile. L'operazione potrebbe fallire però nel caso di implementazioni di `Queue` con dimensione limitata. In particolare il metodo `offer()` inserisce un elemento ritornando `true` o `false` qualora l'operazione di inserimento riesca oppure no. La differenza essenziale con il metodo `add()` (già definito nell'interfaccia `Collection`) è che quest'ultimo può fallire nell'aggiungere un elemento solo lanciando una unchecked exception. Il metodo `offer()` è invece progettato per essere utilizzato quando il fallimento di un inserimento non rappresenta un evento eccezionale, per esempio proprio nel caso di code con dimensione massima.

I metodi `remove()` e `poll()` ritornano e rimuovono l'elemento che si trova in testa alla coda. Nel caso in cui non ci sia niente da rimuovere nella coda, il metodo `poll()` ritorna `null`, mentre `remove()` lancia un'eccezione. Il metodo `poll()` ritorna un riferimento all'oggetto rimosso in caso di successo. I metodi `element()` e `peek()` invece ritornano ma non rimuovono l'elemento che si trova in testa alla coda. Nel caso in cui non ci sia niente da rimuovere nella coda, il metodo `peek()` ritorna `null`, mentre `element()` lancia un'eccezione. Il metodo `peek()` ritorna un riferimento all'oggetto rimosso in caso di successo. La testa della coda è definita dall'implementazione della coda. Esistono **code LIFO** (che sta per *Last In First Out)* che definiscono come testa della coda l'ultimo elemento inserito. Un'implementazione di coda LIFO l'abbiamo già vista: la classe `LinkedList` che mette a disposizione i metodi `addLast()`, `getLast()` e `removeLast()`. `LinkedList` implementa anche l'interfaccia `Deque` e quindi può essere utilizzata come **coda FIFO** (che sta per *First In First Out)* dove la testa della coda è il primo elemento inserito. Infatti mette a disposizione anche i metodi `addFirst()`, `getFirst()` e `removeFirst()`.

L'interfaccia `Deque` è una collezione lineare che supporta l'inserimento e la rimozione dei suoi elementi ad entrambe le estremità. Può essere quindi utilizzata come una coda FIFO o LIFO a seconda della necessità. Come per `Queue`, le funzionalità di questa interfaccia sono duplicate per lanciare eccezioni o ritornare valori speciali. La seguente tabella riassume i principali metodi definiti in questa interfaccia:

Tipo di operazione	Lancia eccezione	Ritorna elemento	Ritorna booleano
Inserimento	`addFirst(e)` `addLast(e)`		`offerFirst(e)` `offerLast(e)`
Rimozione	`removeFirst()` `removeLast()`	`pollFirst()` `pollLast()`	`removeFirstOccurence()` `removeLastOccurence()`
Recupero	`getFirst()` `getLast()`	`peekFirst()` `peekLast()`	

17.5.1 L'interfaccia `BlockingQueue` e implementazioni di `Queue` e `Deque`

Un'implementazione di `Queue` è definita dalla classe `PriorityQueue`, che ordina i propri elementi a seconda del proprio ordinamento naturale (definito mediante l'implementazione dell'interfaccia `Comparable`) o a seconda di un oggetto `Comparator` associato al momento della creazione. Attenzione che usando un iteratore per eseguire cicli su di essa, non è garantito che i suoi elementi siano iterati nell'ordine che ci si aspetta. Infatti, per ragioni prestazionali, i suoi elementi sono gestiti in background senza rispettare l'ordinamento. Il consiglio per iterare gli elementi ordinati è quello di usare uno statement come il seguente:

```
Arrays.sort(pq.toArray())
```

Dove `pq`, è un reference a un oggetto `PriorityQueue`, su cui è invocato il metodo **toArray().** Infatti abbiamo prima trasformato in array la priority queue, e poi ordinato i suoi elementi mediante il metodo statico `sort()` della classe di utilità `Arrays` (che tratteremo nel paragrafo 17.7.2).

Nel package `java.util.concurrent` viene definita anche l'interfaccia `BlockingQueue`. L'interfaccia `BlockingQueue` estende `Queue` e definisce nuovi metodi. Tali metodi bloccano con un thread apposito il recupero e la rimozione di un elemento fino a quando la coda diventa non vuota, e bloccano un inserimento fino a quando lo spazio nella coda bloccante diventa disponibile. Infatti è possibile limitare la capacità di una `BlockingQueue`, di solito mediante un costruttore (come accade per l'implementazione `ArrayBlockingQueue`).

Se non si limita esplicitamente la capacità di una `BlockingQueue`, la capacità massima sarà pari al valore di `Integer.MAX_VALUE`, ovvero il più grande numero intero. In questo caso si parla di "unbounded queue", in italiano "coda illimitata".

Oltre ai metodi che sono ereditati dall'interfaccia `Queue`, e a quelli che bloccano l'inserimento e la rimozione degli elementi secondo quanto detto prima, `BlockingQueue` definisce anche metodi che bloccano queste operazioni per un tempo massimo specificato (time-out). Nel caso il tempo specificato passi prima che l'operazione di inserimento o rimozione sia possibile, viene restituito un valore booleano `false`. Segue una tabella esplicativa:

Tipo di operazione	Lancia eccezione	Ritorna valore speciale	Blocca	Time out
Inserimento	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Rimozione	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Recupero	`element()`	`peek()`		

Per i metodi che gestiscono il timeout bisogna specificare come secondo parametro un `long` (`time` nella tabella) che rappresenta il tempo massimo per eseguire l'operazione. L'unità di tempo viene però specificata con il terzo parametro (`unit` nella tabella) che è di tipo `TimeUnit`. Si tratta di un'enumerazione che definisce come suoi elementi unità temporali: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `NANOSECONDS`, `SECONDS` e `MINUTES`.

Le implementazioni dell'interfaccia `BlockingQueue` sono diverse:

- `LinkedBlockingQueue`, una coda FIFO illimitata (ma che può essere limitata tramite costruttore) e bloccante che in background usa `LinkedList`.
- `PriorityBlockingQueue`, che è una `PriorityQueue` illimitata e bloccante.

- `ArrayBlockingQueue`, una coda limitata che usa in background array.
- `DelayQueue`, una coda illimitata nella quale un suo elemento è reso disponibile solo dopo che un certo timeout configurato è scaduto.
- `SynchronousQueue`, è una coda bloccante nella quale ogni operazione di inserimento deve aspettare la corrispondente operazione di rimozione da un altro thread e viceversa.
- `LinkedTransferQueue`, che implementa la sottointerfaccia di `BlockingQueue`, `TransferQueue`, introdotta con Java 7. Questa permette al codice che aggiunge un elemento alla coda di avere l'opzione per bloccare il codice di un altro thread che vuole usare quell'elemento.

Per quanto riguarda le implementazioni di `Deque`, esse sono molto simili a quelle di `Queue`, tranne per il fatto che implementano `Deque`. La più famosa come abbiamo detto è `LinkedList`. Altre implementazioni da menzionare sono:

- `ArrayDeque`, che è la versione ridimensionabile di `LinkedList` basata in background su array (mentre `LinkedList` usa `List`). Essa è più performante di `LinkedList`, consuma meno memoria e non ammette elementi `null`.
- `LinkedBlockingDeque`, fa parte del package `java.util.concurrent` ed è la versione bloccante della `Deque`. Se la `Deque` è vuota, allora una chiamata a `takeFirst()` o `takeLast()` prima aspetterà che l'elemento sia disponibile, quindi tornerà un reference ad esso e infine lo rimuoverà.

17.6 Le interfacce `Map` e `SortedMap`

Una **mappa** è un oggetto che mappa chiavi a valori, e che non può contenere chiavi duplicate. È definita usando due tipi parametro, una per le chiavi (`K`, inziale di "key" che in inglese significa "chiave") e una per i valori (`V`, iniziale di "value" che in inglese significa "valore"):

```
public interface Map<K,V> { //...
```

Seguono i metodi fondamentali definiti da questa interfaccia:

- `V put(K key, V value)` aggiunge una coppia chiave-valore alla mappa e ritorna il vecchio valore associato alla chiave (se era presente), oppure `null`.
- `V get(Object key)` restituisce il valore associato alla chiave passata in input. Può tornare `null` se la chiave non esiste (oppure se l'implementazione della mappa supporta valori nulli e alla chiave è associato un valore `null`).
- `V remove(Object key)` rimuove il mapping chiave-valore associato alla chiave specificata e ritorna il vecchio valore associato alla chiave (se era presente), oppure `null`.
- `void putAll(Map<? extends K, ? extends V> m)` aggiunge tutte le mappature della mappa passata in input alla mappa corrente.
- `boolean containsKey(Object key)` restituisce `true` o `false` a seconda se la chiave specificata è presente.
- `boolean containsValue(Object value)` restituisce `true` o `false` a seconda se il valore specificato è presente.
- `Set<K> keySet()` ritorna un `Set` contenente tutte le chiavi presenti nella mappa.
- `Collection<V> values()` ritorna una `Collection` di tutti i valori presenti nella mappa.
- `Set<Map.Entry<K, V>> entrySet()` ritorna un `Set` contenente oggetti di tipo `Map.Entry` (ovvero di tipo `Entry`, interfaccia innestata di `Map`) che rappresentano le corrispondenze chiave-valore presenti. Questi oggetti poi mettono a disposizione metodi come `getKey()` e `getValue()`.

Inoltre sono definiti diversi metodi di default tra cui il metodo `forEach()`.

L'interfaccia `SortedMap` è la versione ordinata di `Map` (come `SortedSet` lo è di `Set`), mantiene un ordinamento sulle chiavi, e definisce metodi come i

seguenti:

- `K firstKey()` restituisce la prima chiave della mappa.
- `K lastKey()` restituisce l'ultima chiave della mappa.
- `Comparator<? super K> comparator()` restituisce l'oggetto `Comparator` utilizzato per ordinare gli elementi della mappa.
- `SortedMap<K, V> subMap(K fromKey, K toKey)` restituisce una sottomappa che va dalla chiave `fromKey` alla chiave `toKey`. Questo è possibile solo perché `SortedMap` è ordinata.
- `SortedMap<K, V> headMap(K toKey)` restituisce una sottomappa che va dalla prima chiave alla chiave `toKey`.
- `SortedMap<K, V> tailMap(K fromKey)` restituisce una sottomappa che va dalla chiave `fromKey` specificata all'ultima chiave.

17.6.1 Implementazioni di `Map` e `SortedMap`

Una classe non sincronizzata (e quindi ancora più performante) e del tutto simile ad `Hashtable` (cfr. appendice on line O) è la classe `HashMap`. A meno che non ci sia la necessità di avere una mappa che gestisca la concorrenza in scrittura, `HashMap` dovrebbe sempre essere preferita a `Hashtable`.

Un'altra classe molto simile, ma questa volta implementazione di `SortedMap`, è invece `TreeMap`. La classe `HashMap` è notevolmente più performante rispetto a `TreeMap`. Quest'ultima però gestisce l'ordinamento, che non è una caratteristica di tutte le mappe. Si noti che l'ordinamento è gestito sulle chiavi, non sui valori. Solamente dalla versione 1.6 `TreeMap` è stata rivista per implementare una nuova interfaccia: `NavigableMap`. Questa estende `SortedMap` aggiungendo nuovi metodi come `lowerEntry()`, `floorEntry()`, `ceilingEntry()` e `higherEntry()`, che restituiscono oggetti di tipo `Map.Entry` associati con le chiavi rispettivamente minore, minore o uguale, maggiore o uguale e maggiore della chiave specificata come argomento.

Ricordiamo che l'interfaccia innestata `Map.Entry` astrae una

coppia di tipo chiave-valore che rappresenta un elemento di una mappa.

Inoltre vengono definiti anche i metodi `firstEntry()` e `lastEntry()` che restituiscono rispettivamente il primo e l'ultimo (secondo l'ordinamento) oggetto `Map.Entry` della mappa. Equivalentemente i metodi `pollFirstEntry()` e `pollLastEntry()` restituiscono e rimuovono rispettivamente il primo e l'ultimo (secondo l'ordinamento) oggetto `Map.Entry`.

Ai metodi `keySet()` ed `entrySet()` ereditati da `SortedMap`, che restituiscono rispettivamente un insieme ordinato (`SortedSet`) delle chiavi e degli elementi della mappa, vengono aggiunti i metodi `descendingKeySet()` e `descendingEntrySet()`. Questi restituiscono rispettivamente un insieme (`SortedSet`) delle chiavi e degli elementi della mappa, questa volta ordinato al contrario, dall'ultimo elemento al primo.

I metodi `entrySet()` e `descendingEntrySet()` restituiscono un insieme di valori ordinati per chiave e non per valore.

Esiste anche la classe `LinkedHashMap`, e come possiamo aspettarci è la versione linked di `HashMap`. Possiamo asserire brevemente che le tre implementazioni di riferimento di `Map, HashMap, TreeMap` e `LinkedHashMap` hanno funzionalità e prestazioni equivalenti alle omologhe implementazioni di riferimento di `Set: HashSet, TreeSet` e `LinkedHashSet`.

Infine presentiamo l'interfaccia `ConcurrentMap`, che estende l'interfaccia `Map` e appartiene al package `java.util.concurrent`. Essa definisce metodi come:

- `putIfAbsent(Object key)` che aggiunge una coppia chiave-valore se e solo se la chiave non è già presente.
- `remove(Object key, Object value)` che rimuove una coppia chiave-valore se c'è una corrispondenza nella mappa di una coppia identica.

- `boolean replace(K key, V oldValue, V newValue)` sostituisce con il valore `newValue`, il valore `oldValue`, se e solo se questo è associato alla chiave `key`.
- Tutte le azioni di inserimento e di rimozione degli elementi sono thread-safe e le sue implementazioni, come `ConcurrentHashMap`, sono adatte per sistemi dove c'è l'esigenza di avere mappe condivise da più thread che fanno operazioni su di esse.

17.7 Algoritmi e utilità

Le classi `Collections` ed `Arrays` offrono una serie di metodi statici che rappresentano utilità ed algoritmi utili per le collection.

17.7.1 `Collections`

Esistono diverse tipologie di metodi di utilità definite dalla classe `Collections`. I *metodi wrapper* per collezioni immutabili, quelli che abbiamo definito *sincronizzatori*, che permettono di creare reference per agire sulle collezioni in maniera thread-safe, e poi tanti altri metodi di utilità detti *metodi di convenienza*.

17.7.1.1 Metodi wrapper

In particolare la classe `Collections` offre particolari metodi (detti **metodi wrapper**) per operare su collezioni in modo tale da modificare il loro stato. In realtà tali metodi prendono in input una collezione e ne restituiscono un'altra con le proprietà cambiate. Per esempio i seguenti metodi:

- `public static Collection unmodifiableCollection(Collection c)`
- `public static Set unmodifiableSet(Set s)`
- `public static List unmodifiableList(List list)`
- `public static Map unmodifiableMap(Map m)`
- `public static SortedSet unmodifiableSortedSet(SortedSet s)`

- `public static SortedMap unmodifiableSortedMap(SortedMap m)`

restituiscono **copie immutabili** di ogni collection. Conviene utilizzare collection immutabili tutte le volte che sappiamo che gli elementi della collection non devono cambiare. Infatti una collection immutabile avrà prestazioni migliori di una mutabile, oltre ad essere thread-safe. Ovvero più thread potranno accedere agli elementi della collection senza poterli modificare, e quindi tutti i thread vedranno sempre gli stessi dati. Infatti, queste collection wrapper sono dette immutabili perché se si prova a modificare la collezione viene sollevata una `UnsupportedOperationException`. Per esempio con il seguente codice creiamo una lista wrapper immutabile della variabile `arrayList`:

```
List<String> arrayList = new ArrayList<>();
arrayList.add("a");
arrayList.add("b");
arrayList.add("c");
List<String> wrapperList =
Collections.unmodifiableList(arrayList);
```

Se proviamo ad aggiungere un elemento tramite la lista wrapper:

```
wrapperList.add("d");
```

otteniamo il seguente output:

```
java.lang.UnsupportedOperationException thrown:
        at Collections$UnmodifiableCollection.add
          (Collections.java:1056)
```

quindi il reference `wrapperList` rappresenta una lista non modificabile.

Si faccia attenzione che, una volta creato il reference immutabile `wrapperList`, bisogna utilizzare sempre tale reference. Infatti potremmo sempre modificare la lista utilizzando il reference `arrayList`. Infatti con le seguenti righe:

```
arrayList.add("d");
System.out.println("wrapperList è stata modificata: " +
wrapperList + "!");
```

avremo il seguente output:

```
wrapperList è stata modificata: [a, b, c, d]!
```

Per evitare queste situazioni conviene riassegnare il reference originale alla collezione wrapper in questo modo:

```
arrayList = Collections.unmodifiableList(arrayList);
```

in questo modo non andremo incontro a queste spiacevoli sorprese.

> **Si noti che se il reference originale fosse stato dichiarato di tipo** `ArrayList` **invece che di tipo** `List`**, avremmo ottenuto un errore in compilazione. Questo perché il metodo** `unmodifiableList(List list)` **torna un oggetto di tipo** `List`**, non** `ArrayList`.

Purtroppo non c'è altro modo per modificare la collection! Infatti potremmo avere casi in cui la collection di partenza, invece di contenere oggetti di tipo `String` (che come sappiamo sono immutabili), contiene oggetti non immutabili. In questi casi la struttura degli elementi mutabili di questa collection potrebbe essere comunque modificata, e questo non è detto sia sempre il comportamento voluto. Per esempio consideriamo questa semplice classe di test:

```
public class OggettoModificabile {
    private String nome;
    public OggettoModificabile(String nome) {
        this.nome = nome;
    }
    public void setTitolo(String nome) {
        this.nome = nome;
    }
    public String getTitolo() {
        return nome;
```

```
    }
    public String toString() {
        return nome;
    }
}
```

Se creiamo una lista contenente questi oggetti:

```
List<OggettoModificabile> arrayList = new ArrayList<>();
OggettoModificabile o1 = new OggettoModificabile("NOME_1");
arrayList.add(o1);
arrayList = Collections.unmodifiableList(arrayList);
System.out.println("arrayList prima: " + arrayList);
arrayList.get(0).setTitolo("NOME_2"); //equivalente a
o1.setTitolo("NOME_2");
System.out.println("arrayList dopo: " + arrayList);
```

otterremo il seguente output:

```
arrayList prima: [NOME_1]
arrayList dopo: [NOME_2]
```

Quindi le copie immutabili create tramite i metodi wrapper di `Collections`, sono realmente immutabili se e solo se anche gli elementi della collezione sono immutabili, e il reference originale della collezione è riassegnato al risultato del metodo wrapper.

17.7.1.2 Metodi sincronizzatori

I seguenti metodi invece:

- `public static Collection synchronizedCollection(Collection c)`
- `public static Set synchronizedSet(Set s)`
- `public static List synchronizedList(List list)`
- `public static Map synchronizedMap(Map m)`
- `public static SortedSet synchronizedSortedSet(SortedSet s)`
- `public static SortedMap synchronizedSortedMap(SortedMap m)`

sincronizzano tutte le tipologie di collection.
Per esempio, anche l'`ArrayList` si può sincronizzare sfruttando uno dei tanti metodi statici di utilità della classe `Collections` (da non confondere con `Collection`):

```
List<String> arrayList = new ArrayList<>();
arrayList.add("a");
arrayList.add("b");
arrayList.add("c");
List<String> synchList =
Collections.synchronizedList(arraylist);
```

Anche in questo caso, se utilizziamo il reference originale la collezione non sarà thread-safe. Come nel caso dei metodi wrapper bisogna quindi riassegnare il reference originale al risultato del metodo sincronizzatore.

Sincronizzare una collection non significa sincronizzare anche il rispettivo `Iterator`. È quindi obbligatorio eseguire un ciclo con un `Iterator` quantomeno all'interno di un blocco sincronizzato come nel seguente esempio:

```
Collection<String> c =
Collections.synchronizedCollection(synchList);
synchronized(c) {
    Iterator<String> iterator = c.iterator();
    while (iterator.hasNext())
        faQualcosa(iterator.next());
}
```

Questo perché l'iterazione avviene con chiamate multiple sulla collezione che vengono composte in un'unica operazione.

Abbiamo già asserito che in generale le prestazioni di un `Vector` (cfr. appendice O on line) sono inferiori rispetto a quelle di un `ArrayList`. La situazione in questo caso però si capovolge: il `Vector` ha prestazioni superiori a quelle di un

`ArrayList` **sincronizzato.**

Lo stesso discorso vale anche per le mappe con il metodo `synchronizedMap()`. In questo caso l'`Iterator` si estrae dall'insieme delle chiavi e bisogna sincronizzare il blocco di codice sull'oggetto mappa:

```
Map map = Collections.synchronizedMap(myHashMap);
Set set = map.keySet();
synchronized (map) {
    Iterator iterator = set.iterator();
    while (iterator.hasNext())
        faQualcosa (iterator.next());
}
```

Oltre ai metodi wrapper, `Collections` ha metodi che implementano complicati algoritmi per gli oggetti `List` come `sort()` (ordina con l'algoritmo "merge sort"), `shuffle()` (mischia gli elementi della lista specificata, il contrario di ordina), `max()` (restituisci l'elemento massimo), `min()` (restituisci l'elemento minimo), `reverse()` (inverti l'ordine degli elementi), `rotate()` (ruota tutti gli elementi della lista specificata, del numero di posizioni specificato come secondo parametro), `swap()` (scambia gli elementi agli indici specificati), `copy()` (copia una lista in un'altra) e altri ancora.

17.7.1.3 Metodi di convenienza

Altri **metodi detti di convenienza** consentono la creazione di collection immutabili di un numero definito di oggetti identici col metodo `nCopies()`, o di un oggetto singleton (istanziabile una sola volta) ottenibile mediante il metodo `singleton()` (che restituisce un `Set` che fa da wrapper immutabile all'input specificato). Per esempio, potremmo rimuovere tutti gli elementi di un certo tipo da una collection sfruttando il metodo `removeAll()` di `Collection` e il metodo `singleton()` di `Collections`:

```
collection.removeAll(Collections.singleton(oggettoDaRimuovere))
```

Con un altro esempio è possibile eliminare tutti gli oggetti `null` da una

collection:

```
collection.removeAll(Collections.singleton(null));
```

17.7.2 `Arrays`

La classe `Arrays` contiene alcuni dei metodi-algoritmi di `Collections` ma relativi ad array, come `sort()`. Inoltre possiede il metodo `asList()`, che può trasformare un array in una `List` (precisamente in un oggetto di tipo `Arrays.ArrayList`). Per esempio:

```
List<String> l = Arrays.asList(new String[] {"1", "2",
"3"});
```

crea un'implementazione di `List` a partire dall'array specificato al volo come parametro (di cui non rimarrà nessun puntamento e quindi verrà poi reso disponibile al Garbage Collector per la dismissione).

Con le versioni 5 e 6 di Java sono stati aggiunti nuovi metodi molto interessanti per la classe `Arrays`. Per esempio, dalla versione 5 esistono nove versioni del metodo `toString(array)` soggetto a overload per prendere in input tutte le possibili combinazioni di array di tipi primitivi e di `Object`. Con essi si potranno stampare all'interno di parentesi quadre tutti gli elementi di un array (separati da virgole) senza ricorrere ad un ciclo. La versione 6 invece ha introdotto nove versioni del metodo `copyOfRange(array, from ,to)`, per creare nuovi sotto-array a partire da array di dimensioni più grandi.

Infine la versione 8 ha introdotto il metodo `parallelSort()` che sfruttando l'algoritmo definito dal framework fork/join, ordina in maniera parallela un array, con prestazioni superiori rispetto a `sort()` nel caso di liste sufficientemente lunghe.

17.7.3 Metodi di convenienza per creare collection immutabili

Java 9 introduce una nuova sintassi per la creazione di mappe, insiemi e liste immutabili. Infatti sono stati introdotti nelle interfacce `Map`, `Set` e `List`, dei **metodi statici di convenienza** che facilitino la creazione di collezioni immutabili come visto per

i metodi wrapper di Collections nel paragrafo 17.7.1.1. Dichiarando meno oggetti questa nuova sintassi permette di ottimizzare l'utilizzo della memoria. Si tratta di metodi factory di tipo `of()`, simili ai metodi `of()` che abbiamo visto nella libreria Date-Time API.

17.7.3.1 Interfaccia `Set`

Iniziamo a parlare dell'interfaccia `Set` considerando le seguenti righe di codice con cui creiamo un `Set` immutabile di stringhe:

```
Set<String> immutableSet = new HashSet<>();
immutableSet.add("a");
immutableSet.add("b");
immutableSet.add("c");
immutableSet = Collections.unmodifiableSet(immutableSet);
```

Lo stesso risultato si può ottenere con meno righe di codice utilizzando il metodo `asList()` della classe `Arrays`:

```
Set<String> immutableSet = new HashSet<>(Arrays.asList("a",
"b", "c"));
immutableSet = Collections.unmodifiableSet(immutableSet);
```

Con Java 9 possiamo utilizzare una sintassi ancora meno verbosa, sfruttando il metodo `of()` dell'interfaccia `Set`:

```
Set<String> immutableSet = Set.of("a", "b", "c");
```

Ricordiamo che se provassimo a modificare la collezione sarà sollevata una `UnsupportedOperationException`. Infatti ecco l'output su JShell di un'istruzione che prova ad aggiungere un nuovo elemento alla collezione:

```
jshell> immutableSet.add("d");
|  java.lang.UnsupportedOperationException thrown:
|        at ImmutableCollections.uoe
(ImmutableCollections.java:71)
|        at ImmutableCollections$AbstractImmutableSet.add
           (ImmutableCollections.java:281)
```

In un `Set` non è possibile aggiungere due elementi duplicati, quindi il metodo `of()` lancerebbe una `IllegalArgumentExeption` se gli fosse passata una

collezione con elementi duplicati. Ecco un esempio eseguito con JShell:

```
jshell> Set<String> immutableSet = Set.of("a", "a", "c");
|  java.lang.IllegalArgumentException thrown: duplicate
element: a
|        at ImmutableCollections$SetN.<init>
           (ImmutableCollections.java:463)
|        at Set.of (Set.java:501)
```

Il metodo `of()` nell'interfaccia `Set` ha vari overload in modo tale da accomodare un qualsiasi numero di elementi da aggiungere.

17.7.3.2 Interfaccia `Map`

Per l'interfaccia `Map` vale più o meno quanto detto per `Set`. Per esempio il seguente codice:

```
Map<String, Integer> immutableMap = new HashMap<Integer,
String>();
immutableMap.put(1, "a");
immutableMap.put(2, "b");
immutableMap.put(3, "c");
immutableMap = Collections.unmodifiableMap(immutableMap);
```

con Java 9 si può riscrivere come:

```
Map immutableMap = Map.of(1, "a", 2, "b", 3, "c");
```

Si noti l'alternanza tra chiave e valore dei parametri, da utilizzare per il metodo `of()` di `Map`. Una differenza evidente rispetto al metodo `of()` definito nell'interfaccia `Set`, consiste nel fatto che se dobbiamo creare una mappa immutabile che abbia più di dieci coppie chiave-valore, allora dobbiamo utilizzare il metodo `ofEntries()` in luogo del metodo `of()`, con la seguente sintassi:

```
Map <Integer, String> immutableMap = Map.ofEntries(
Map.entry(1, "a"),
Map.entry(2, "b"),
Map.entry(3, "c"),
//...
Map.entry(21, "z"));
```

Infatti il metodo `of()` ha diversi overload sino ad accettare al massimo dieci coppie chiave-valore, mentre `ofEntries()` ha come parametro un varargs di `Map.Entry`. Come nel caso dell'interfaccia `Set`, il metodo `of()` di `Map` lancerà una `IllegalArgumentExeption` se gli sarà passata una collezione con chiavi duplicate:

```
jshell> Map immutableMap = Map.of(1, "a", 2, "b", 2, "c");
|  java.lang.IllegalArgumentException thrown: duplicate key: 2
|        at ImmutableCollections$MapN.<init>
            (ImmutableCollections.java:681)
|        at Map.of (Map.java:1348)
```

> **Per una mappa l'ordine degli elementi non dipende dall'ordine in cui sono stati inseriti.**

17.7.3.3 Interfaccia `List`

Per l'interfaccia `List` la situazione è più semplice visto che accetta tranquillamente duplicati, e il suo metodo `of()` è ricco di overload in modo tale da accettare qualsiasi numero di elementi da aggiungere:

```
List<String> immutableList = Arrays.asList("a", "b", "c");
immutableList = Collections.unmodifiableList(immutableList);
```

Che con Java 9 si può riscrivere come:

```
List immutableList = List.of("a", "b", "c");
```

> **Essendo questa nuova sintassi equivalente all'utilizzo dei metodi wrapper di Collections, sussistono le medesime possibilità di modificare la collezione se gli oggetti contenuti in essa non sono immutabili (cfr. paragrafo 17.7.1.1).**

17.8 Introduzione alla libreria Stream API

La programmazione in generale si sta evolvendo per soddisfare le nuove

architetture hardware dei nostri giorni. I programmatori hanno sempre più spesso a che fare con ciò che viene definito “Big Data” (Terabyte di dati), con CPU multicore, cluster di computer in cloud e così via. Questo spesso significa usare il paradigma della programmazione parallela. Linguaggi moderni come Groovy e Scala, e framework come il MapReduce di Google hanno da subito puntato molto sulle nuove architetture. Java ha quindi definito la libreria di alto livello **Stream API** che semplifica enormemente la programmazione Java (in particolare relativamente all’uso delle collezioni) e contemporaneamente offre supporto alla programmazione parallela. Tutto gira intorno alla definizione di `Stream`, una nuova interfaccia definita nel package `java.util.stream`.

Uno **stream** rappresenta un flusso che ha come sorgente una collezione di dati, e su cui si possono eseguire operazioni complicate con uno sforzo di programmazione minimo, spesso basato su espressioni lambda e reference a metodi. Tipicamente uno stream viene creato a partire da una collection tramite la chiamata al metodo `stream()`, ma ci sono diversi modi per ottenere oggetti `Stream`. Per esempio è possibile istanziare uno `Stream` passando un array al metodo statico `of()`:

```
Stream <String> stringsStream = Stream.of(arrayDiStringhe);
```

oppure passando all’altra versione del metodo `of()` direttamente dei valori:

```
Stream <String> stringsStream = Stream.of("Take", "The",
"Time");
```

o tramite il metodo statico `generate()` della stessa classe `Stream`. In questo caso il metodo `generate()` ci fornisce uno stream infinito perché ogni volta che vogliamo un valore dallo stream sarà creato. Per esempio il seguente frammento di codice istanzia uno `Stream` di infiniti numeri casuali, usando un reference al metodo `random()` della classe `Math` (cfr. paragrafo 12.7) che restituisce un numero casuale di tipo `double` maggiore o uguale di `0.0` e minore di `1`.

```
Stream<Double> randomDoubles =
Stream.generate(Math::random);
```

Inoltre, un’oggetto `Stream` è molto utile per iterare sulle collezioni, mediante

il suo metodo `forEach()`, e da questo sfruttare le cosiddette **operazioni di aggregazione.** Queste ultime vengono utilizzate congiuntamente a espressioni lambda e reference a metodi, rendendo il codice estremamente potente e allo stesso tempo conciso. Queste espressioni infatti, ritornano sempre un altro oggetto di tipo `Stream`, fornendo quindi la possibilità di concatenare le chiamate a queste funzioni. Queste concatenazioni di chiamate sono definite comunemente **pipeline.** Per esempio, considerando l'oggetto `smartphones` dei paragrafi precedenti, con il seguente statement:

```
smartphones.stream().filter(s-
>"Samsung".equals(s.getMarca())).
  forEach(s->System.out.println(s));
```

stamperemo:

```
Samsung Note 74
```

Infatti, abbiamo chiamato il metodo `stream()` sulla collection `smartphones`, ottenendo appunto un primo oggetto `Stream`. Su di esso abbiamo invocato il metodo `filter()` che prende in input un `Predicate` (cfr. paragrafo 16.3.1) che, ricordiamo, era usato proprio per implementare dei test che ritornano un booleano. Quindi gli abbiamo passato un'espressione lambda che implementa un controllo con il metodo `equals()` sulla marca dell'elemento. Il metodo `filter()` ritorna nuovamente un'oggetto `Stream` relativo alla collezione filtrata, che conterrà un unico elemento (visto che nella collezione c'è un unico smartphone di marca Samsung). Su questo stream invochiamo il metodo `foreach()` della classe `Stream`, che esegue un'iterazione completa sugli elementi della collection, e permette di eseguire un'operazione su ogni elemento della stessa. Anche in questo caso ci limitiamo a stampare gli elementi.

> **L'operazione `filter()` viene detta "operazione di aggregazione" perché progettata per restituire un nuovo stream, su cui invocare altre operazioni.**

17.8.1 Definizioni di `Stream` e pipeline

Solitamente usiamo le collection per raggruppare elementi in un solo oggetto, per poi eseguire operazioni sui loro elementi all'interno di cicli. Uno **stream** (che in italiano possiamo tradurre come **flusso)** è una sequenza di elementi su cui possiamo iterare. L'iterazione è unica e una volta terminata, lo stream non è più utilizzabile. A differenza di una collection, uno stream non immagazzina elementi, ma permette di operare su questi elementi attraverso pipeline. Questo ha il grande vantaggio di consentirci di eseguire tali operazioni in modo molto conciso ed efficiente. Il vantaggio più evidente è quando concateniamo le operazioni da fare sugli elementi della collezione tramite pipeline. Possiamo definire **pipeline** (che in italiano possiamo tradurre come **conduttura** inteso come "tubo composto da diversi tubi messi in sequenza per far passare un flusso di qualcosa") una sequenza di operazioni di aggregazione. Una pipeline è costituita da tre elementi:

1. una **sorgente** (in inglese **source**): potrebbe essere una collection o un array. Ma esistono anche altri tipi di sorgente, come canali di input/output, il metodo `generate()` della stessa classe `Stream` e altri ancora. Una sorgente dovrebbe mettere a disposizione un metodo per ottenere un oggetto `Stream`;
2. zero o più **operazioni di aggregazione** (oppure **operazioni intermedie**): queste operazioni hanno la caratteristica di ritornare un nuovo oggetto `Stream`. Nell'esempio precedente abbiamo visto come operazione di aggregazione il metodo `filter()`. Altri tipici metodi intermedi sono i metodi `map()`, `mapToDouble()`, etc. Tutti i metodi intermedi sono chiamati solo all'ultimo momento (si dice che sono metodi **lazy**, in italiano "pigri"), nel senso che saranno invocati solo quando sarà invocato il metodo terminale, che ha bisogno dei metodi intermedi per poter essere eseguito;
3. una **operazione terminale:** un metodo che restituisce un risultato che non è un oggetto `Stream`. Nell'esempio precedente abbiamo visto come operazione terminale il metodo `forEach()`.

La terminologia inglese parla metaforicamente di flussi e tubi. Gli stream (i flussi) non immagazzinano dati ma

trasportano gli elementi dalla sorgente (nel nostro caso una collection) attraverso la tubatura (la pipeline), trasformando (filtrando, modificando, etc.) il risultato finale. Questo tipo di astrazione è simile a quella che da sempre viene proposta per la libreria dell'Input-Output tramite `InputStream` **e** `OutputStream` **come vedremo nel prossimo capitolo.**

Per esempio, sempre considerando la classe `Smartphone` di prima, e supponendo che il metodo `getSmartphones()` ritorni una collezione di oggetti `Smartphone`, allora potremmo calcolare la media del prezzo degli smartphone di marca Samsung con il seguente codice:

```
Collection<Smartphone> smartphones = getSmartphones();
double average = smartphones.stream().
  filter(s -> s.getMarca().equals("Samsung")).
  mapToDouble(Smartphone::getPrezzo).average().getAsDouble();
```

Si noti che la pipeline specificata concatena ben cinque metodi. Dalla sorgente `smartphones` otteniamo prima uno `Stream` invocando il metodo `stream()`. Su questo stream andiamo a chiamare il metodo `filter()` a cui passiamo un'espressione lambda per filtrare tutti i modelli di marca Samsung. Questo metodo restituisce a sua volta uno `Stream` su cui chiamiamo il metodo `mapToDouble()` a cui passiamo un reference al metodo `getPrezzo()` della classe `Smartphone`. Il metodo `mapToDouble()` restituirà uno stream (in questo caso di tipo `DoubleStream` ovvero uno stream che contiene solo valori `Double`) su una collezione di oggetti `Double` con i prezzi degli smartphone Samsung filtrati. Su quest'ultimo stream chiamiamo il metodo `average()`, che esegue un media matematica degli elementi della collection di `Double` e ritorna un oggetto di tipo `OptionalDouble` (cfr. prossimo paragrafo). Infine invochiamo il metodo `getAsDouble()` che ritorna il double incapsulato nell'oggetto `OptionalDouble`.

Un metodo come `average()` **viene detto "reduction**

operation” (in italiano “operazione di riduzione”), in quanto ritorna un unico valore calcolato da più elementi dello stream.

17.8.2 Classi Optional

`OptionalDouble` è una delle importanti classi di Java dette *classi optional*. Le **classi optional** sono delle classi wrapper che consentono di evitare `NullPointerException`. Queste infatti possono immagazzinare valori di un certo tipo, e sfruttare i propri metodi affinché l’eccezione più famosa non sia lanciata, senza dover ricorrere ai soliti controlli di nullità. Oltre ad `OptionalDouble`, esistono le classi `OptionalInt`, `OptionalLong` e `Optional<T>`. Per capire come funzionano tali classi supponiamo di avere il seguente metodo:

```
public static String getTitoloMaiuscolo(String titolo) {
    if (titolo != null) {
        return titolo.toUpperCase();
    }
    return "NESSUN TITOLO";
}
```

Questo metodo restituisce la stringa parametro `titolo` in maiuscolo, oppure la stringa `NESSUN TITOLO` nel caso la stringa `titolo`, sia `null`. Per evitare di incorrere in una `NullPointerException`, abbiamo dovuto fare dei verbosi controlli di nullità all’interno del nostro metodo. In compenso il metodo sembra robusto.

Infatti eseguendo le seguenti righe di codice di test:

```
System.out.println(getTitoloMaiuscolo("titolo"));
System.out.println(getTitoloMaiuscolo(null));
```

otterremo il seguente output:

```
TITOLO
NESSUN TITOLO
```

Per evitare la verbosità possiamo utilizzare un oggetto `Optional` ridefinendo il metodo in questo modo:

```
public static String getTitoloMaiuscoloOpt(String titolo) {
    Optional<String> opt = Optional.ofNullable(titolo);
    return opt.orElse("NESSUN TITOLO").toUpperCase();
}
```

Il metodo `ofNullable()` restituisce un oggetto `Optional` che contiene l'oggetto `titolo`. Se `titolo` è `null` l'oggetto `Optional` sarà comunque creato con un contenuto `null`. Il metodo `orElse()` invece restituisce la stringa in input (in questo caso la stringa `NESSUN TITOLO`) nel caso il parametro `titolo` sia `null`. L'utilità delle classi optional è massimizzata se usata in concomitanza di pipeline di stream. Infatti hanno dei metodi a cui è possibile passare espressioni lambda o valori di ritorno alternativi. Si noti che, nell'esempio sugli smartphone presentato nel precedente paragrafo, se il metodo `getSmartphones()` non avesse restituito un oggetto smartphone di marca Samsung, avremmo avuto il seguente output:

```
Exception in thread "main" java.util.NoSuchElementException:
  No value present
    at
java.util.OptionalDouble.getAsDouble(OptionalDouble.java:118)
    at TestPipeline.main(TestPipeline.java:11)
```

Quindi, qual è il vantaggio di avere una `NoSuchElementException` in luogo di una `NullPointerException`? Il metodo `orElse()`, come abbiamo visto, permette di specificare un elemento alternativo nel caso l'oggetto `Optional` sia vuoto. Possiamo quindi modificare il metodo in questo modo:

```
Optional<Smartphone> found = smartphones.stream().
filter(s -> s.getMarca().equals("Samsung")).findFirst();
System.out.println(found.orElse(new Smartphone("Samsung",
"Note 74")));
```

Nell'esempio abbiamo usato il metodo `findFirst()`, un metodo terminale che torna un oggetto `Optional` con il primo elemento dello stream, oppure un `Optional` vuoto. Prima di stampare invochiamo su di esso il metodo `orElse()` a cui passiamo un valore alternativo. Questo sarà stampato nel caso l'`Optional` sia vuoto. Abbiamo evitato così i soliti verbosi controlli di nullità.

17.8.2.1 Altri metodi `or()`

`Optional` dichiara anche il metodo `orElseGet()`. Questo è molto simile al metodo `orElse()` e si comporta allo stesso modo nel caso l'elemento dell'oggetto `Optional` sia vuoto. La differenza è che l'input di questo metodo è di tipo `Supplier`, (cfr. paragrafo 16.3.3) mentre l'input del metodo `orElse()` è del tipo parametro dell'`Optional`. Le dichiarazioni dei due metodi sono le seguenti:

- `public T orElse(T other)`
- `public T orElseGet(Supplier <? extends T> supplier)`

Per invocare il metodo `orElse()` bisogna per forza istanziare un oggetto. Nel metodo `getTitoloMaiuscoloOpt()` del precedente esempio infatti avevamo istanziato la stringa `NESSUN TITOLO` con il seguente statement:

```
return opt.orElse("NESSUN TITOLO").toUpperCase();
```

Invece per usare `orElseGet()` è necessario passargli in input un'espressione lambda o un reference a un metodo. Nel nostro caso possiamo sostituire questa istruzione con la seguente che utilizza una semplice espressione lambda:

```
return opt.orElseGet(() -> "NESSUN TITOLO").toUpperCase();
```

La differenza è che l'espressione lambda non è invocata se non necessario, quindi la stringa `NESSUN TITOLO` non sarebbe mai istanziata.
Come si può intuire dal nome, il metodo `orElseThrow()` nel caso di nullità dell'oggetto contenuto nell'optional, lancerà un'eccezione. Il metodo è dichiarato in modo da prendere in input un `Supplier` che ritorna un `Throwable`:

```
public <X extends Throwable> T orElseThrow(Supplier <?
extends X> es) throws X
```

Per esempio potremmo utilizzare nell'esempio precedente il seguente codice:

```
return opt.orElseThrow(IllegalArgumentException::new);
```

che genererà il seguente output:

```
titolo
Exception in thread "main" java.lang.IllegalArgumentException
      at
java.base/java.util.Optional.orElseThrow(Optional.java:385)
      at
  TestSimpleOptional.getTitoloMaiuscoloOpt(TestSimpleOptional.ja

      at TestSimpleOptional.main(TestSimpleOptional.java:23)
```

17.8.2.2 Metodi `get()`, `filter()`, `map()` e `flatMap()`

Il metodo `get()` invece, restituirà l'oggetto contenuto nell'optional. Se questo oggetto è `null` però sarà lanciata l'eccezione `NoSuchElementException`. Oltre a `Stream` anche `Optional` dichiara un metodo `filter()` che prende in input un `Predicate`. Esso restituisce un `Optional` completo di valore solo se esso coincide con quanto ritorna il `Predicate` in input, altrimenti ritorna un `Optional` vuoto. Nel caso il valore del `Predicate` sia `null`, allora lancia una `NullPointerException`. Si tratta di un metodo di cui si può apprezzare meglio l'utilità all'interno di pipeline. Il metodo `map()` consente di mappare un primo oggetto ad un secondo oggetto, restituendo un `Optional` contenente il secondo oggetto. Questo è possibile perché il metodo `map()` prende in input una `Function` (cfr. paragrafo 16.3.4), ovvero un'espressione lambda o un reference a metodo che prende in input un tipo (il primo oggetto) e ne restituisce un altro (il secondo oggetto). Il metodo `map()` è definito così:

```
public <U> Optional<U> map(Function<? super T,? extends U>
mapper)
```

Dove `T` è quello che abbiamo definito primo oggetto, e `U` è quello che abbiamo definito secondo oggetto. Nel seguente esempio, definiamo un metodo con un'unica pipeline che restituirà `true`, se l'oggetto in input ha un titolo che contiene la parola `Java`:

```
public static boolean isTitoloJava(OggettoModificabile o) {
    return Optional.ofNullable(o).
      map(OggettoModificabile::getTitolo).
      filter(t -> t.contains("Java")).
      isPresent();
```

```
}
```

Infatti abbiamo creato un metodo a cui viene passato in input un `OggettoModificabile` (la cui classe abbiamo definito nel paragrafo 17.7.1.1). Nella pipeline tale oggetto è dapprima inserito in un oggetto `Optional` tramite il metodo statico `ofNullable()`. In questo modo ci mettiamo al sicuro da eventuali `NullPointerException` visto che tale metodo, nel caso prenda in input un valore `null`, istanzierà comunque un oggetto `Optional` vuoto. Su questo oggetto `Optional` viene invocato il metodo `map()`, che prende in input, come `Function`, il reference a metodo `getTitolo()` della classe `OggettoModificabile`. Si noti come sia stata utilizzata la sintassi che abbiamo definito nel paragrafo 16.2.4 “Reference a un metodo d’istanza di un certo tipo”, ovvero la sintassi del tipo:

```
NomeTipo::nomeDelMetodoDIstanza
```

questo perché non conosciamo il reference dell’oggetto `Optional` su cui stiamo lavorando (quello che ci ha restituito il metodo `ofNullable()`), ma sappiamo che su di esso possiamo invocare il metodo `getTitolo()`. Il metodo `map()` restituisce quindi un oggetto `Optional` contenente il valore del `titolo` dell’oggetto modificabile. Su questo oggetto `Optional` invochiamo il metodo `filter()`, a cui passiamo un’espressione lambda che rappresenta il `Predicate`, che controlla che il titolo contenga la stringa `Java`. Il metodo `isPresent()` restituisce `true` se è invocato su un `Optional` che contiene un valore. Quindi, eseguendo le seguenti righe di codice:

```
OggettoModificabile o = new OggettoModificabile("Manuale di
Java 9");
System.out.println(isTitoloJava(o));
o.setTitolo("SQL");
System.out.println(isTitoloJava(o));
o.setTitolo(null);
System.out.println(isTitoloJava(o));
o = null;
System.out.println(isTitoloJava(o));
```

otterremo il seguente output:

```
true
false
```

```
false
false
```

Infine esiste anche il metodo `flatMap()`, che è così definito:

```
public <U> Optional<U> flatMap(
  Function<? super T,? extends Optional<? extends U>>
mapper)
```

Tale metodo differisce dal metodo `map()` solo per il fatto che la `Function` definisce come tipo di ritorno proprio un oggetto `Optional`. Quindi il metodo `flatMap()` non si occuperà di creare l'`Optional` per contenere il valore della `Function` come fa il metodo `map()`, bensì bisognerà passarglielo già creato. Per esempio, se riscriviamo il metodo `isTitoloJava()` utilizzando il metodo `flatMap()`, ci conviene prima definire un metodo come il seguente:

```
public static Optional<String>
getTitoloOpt(OggettoModificabile o) {
   return Optional.ofNullable(o.getTitolo());
}
```

cherestituisce un`Optional` con tipo `String`, a partire da un`OggettoModificabile`. L'`Optional` conterrà il titolo dell'`OggettoModificabile`. A questo punto possiamo riscrivere il metodo `isTitoloJava()` passando al metodo `flatMap()` il reference del metodo `getTitoloOpt()`:

```
public static boolean isTitoloJavaFM(OggettoModificabile o)
{
    return Optional.ofNullable(o).
   flatMap(TestSimpleOptional::getTitoloOpt).
      filter(t -> t.contains("Java")).
      isPresent();
}
```

17.8.3 Reduction Operations

Tornando agli stream, abbiamo già asserito che il metodo `average()` è considerato una **reduction operation** (in italiano una **operazione di riduzione)** ovvero un metodo che a partire dagli elementi di uno stream, ritorna un solo valore. Altre tipiche implementazioni di operazioni di

riduzione sono metodi come `max()`, `min()`, `sum()` e `count()` i cui nomi sono auto esplicativi. Per esempio con il seguente statement:

```
long count = smartphones.stream().
  filter(s -> s.getMarca().equals("Apple")).count();
```

contiamo i numeri di smartphone di marca Apple. È semplice interpretare questa pipeline: uno stream di smartphone viene filtrato per marca e, sullo stream di smartphone risultante dall'operazione di filtro, è invocato il metodo `count()`, che conta le occorrenze all'interno dello stream. Metodi come quest'ultimo sono **operazioni terminali.** Esistono però anche altre operazioni di riduzione che ritornano collezioni invece che singoli elementi. Stiamo parlando dei metodi `reduce()` e `collect()`.

17.8.3.1 Metodo `reduce()`

Il metodo `reduce()` esiste in più versioni (overload). La più famosa prende in input due parametri:

1. un oggetto detto **identity** che rappresenta sia il valore iniziale sia il valore da restituire nel caso non ci siano elementi nello stream.
2. un oggetto di tipo `BinaryOperator` chiamato **accumulator.** `BinaryOperator` è un'altra interfaccia funzionale che estende `BiFunction` (cfr. paragrafo 16.3.4), dove i due operandi e il tipo di ritorno sono dello stesso tipo.

Per esempio, per recuperare il prezzo totale degli smartphone nella collezione di una certa marca potremmo scrivere:

```
long prezzoTotale = smartphones.stream().
  filter(s -> s.getMarca().equals("Samsung")).
map(Smartphone::getPrezzo).reduce(0, (x,y) -> x+y);
```

anche se sarebbe stato più semplice usare il metodo `sum()` di `Integer`:

```
long prezzoTotale = smartphones.stream().
filter(s -> s.getMarca().equals("Samsung")).
map(Smartphone::getPrezzo).reduce(0, Integer::sum);
```

Il metodo `reduce()` però, esegue un calcolo per ogni coppia e restituisce un

valore, e questo in situazioni più complesse della precedente potrebbe rappresentare un problema per le prestazioni.

17.8.3.2 Metodo `collect()` *e la classe* `Collectors`

Il metodo `collect()` è utilizzato spesso congiuntamente ai metodi statici messi a disposizione dalla classe `Collectors`. La classe `Collectors` contiene metodi come `toList()`, `toSet()`, `toCollection()`, `reducing()`, `joining()` e `groupingBy()`, che consentono di raggruppare risultati dagli stream. Per esempio con il seguente statement:

```
List<String> list1 = smartphones.stream().map(
  Smartphone::getModello).collect(Collectors.toList());
```

otteniamo una lista di stringhe di tutti i modelli della collection di `Smartphone`. Infatti il metodo `toList()`, restituisce una lista di oggetti dello stream. Invece con la seguente istruzione, utilizziamo il metodo `toSet()`:

```
Set<String> list2 =
smartphones.stream().map(Smartphone::getMarca).
  collect(Collectors.toSet());
```

ed otteniamo un `Set` (quindi senza duplicati) di stringhe contenente le marche degli smartphone. Se però vogliamo immagazzinare il risultato in un'implementazione particolare come `TreeSet`, per avere un `Set` ordinato, allora possiamo utilizzare il metodo `toCollection()`:

```
Set<String> list2 =
smartphones.stream().map(Smartphone::getMarca).
  collect(Collectors.toCollection(TreeSet::new));
```

Notiamo che in questo caso abbiamo dovuto passare un reference al metodo costruttore di `TreeSet`, per specificare quale collection utilizzare. Sfruttando la classe `StringJoiner` di `java.util` (che permette di associare un separatore a varie stringhe), il metodo `joining()` di `Collectors` ci consente di stampare la lista dei modelli di smartphone separati da virgole senza più il bisogno di creare algoritmi per capire se bisogna aggiungere o meno il separatore, il codice è banale:

```
String modelliConSeparatore = smartphones.stream().
```

```
    map(Smartphone::toString).collect(Collectors.joining(";
  "));
```

e stamperà:

```
Motorola G55; Samsung Note 74; Apple IPhone 225;
```

È interessante raggruppare gli elementi delle collezioni per marca con il metodo `groupingBy()`:

```
  Map<String, List<Smartphone>> map = smartphones.stream().
    collect(Collectors.groupingBy(Smartphone::getMarca));
```

Si noti come viene restituita una mappa che associa alla marca gli smartphone: uno sforzo davvero minimo rispetto a quello che avremmo dovuto scrivere prima di Java 8! Un'altro interessante utilizzo di `Collectors` sono i **metodi summarizing** (`summarizingInt()`, `summarizingDouble()` e `summarizingLong()`) che permettono di ottenere un oggetto di tipo `DoubleSummaryStatistics`:

```
  DoubleSummaryStatistics stats =
  smartphones.stream().collect(
    Collectors.summarizingDouble(Smartphone::getPrezzo));
```

Quest'oggetto contiene tutte le informazioni ottenibili dalle operazioni di riduzione basilari. Infatti stampando il suo metodo `toString()`, otterremo il seguente output:

```
DoubleSummaryStatistics{count=8, sum=3326,000000,
min=70,000000,
  average=415,750000, max=721,000000}
```

17.8.4 Stream paralleli e prestazioni

In un esempio precedente filtravamo e stampavamo gli elementi `Smartphone` usando una pipeline che usava uno `Stream` come nel codice seguente:

```
  smartphones.stream().filter(s-
  >"Samsung".equals(s.getMarca())).
    forEach(s->System.out.println(s));
```

Allo stesso modo avremmo potuto anche usare uno `Stream` che eseguisse le

stesse operazioni in maniera parallela. Questo si ottiene semplicemente invocando il metodo `parallelStream()` sulla collezione, in luogo del metodo `stream()`. Per il resto non cambia nulla:

```
smartphones.parallelStream().filter(s-
>"Samsung".equals(s.getMarca())).
  forEach(s->System.out.println(s));
```

Questo *stream parallelo*, per filtrare gli elementi della collezione, utilizza un algoritmo di tipo *fork/join*. Tale algoritmo divide la collezione in più parti e assegna l'operazione di filtraggio di ogni parte ad un thread diverso. Quando tutti i thread hanno terminato la loro esecuzione, i risultati sono ricongiunti per ottenere lo stesso risultato che si sarebbe ottenuto filtrando tutta la collezione interamente. Il vantaggio è che i vari thread potrebbero svolgere il lavoro in parallelo se ci sono abbastanza core di CPU a disposizione sulla macchina, riducendo il tempo di esecuzione del processo. L'esempio è puramente didattico in quanto è completamente inutile usare uno stream parallelo su una collezione di soli tre elementi. Inoltre questo tipo di esecuzione parallela ha un costo: quello di dover ogni volta suddividere (fork) la collezione in più collezioni per poi effettuare il ricongiungimento (join). Quindi nella maggior parte dei casi, non bisogna aspettarsi che i tempi di esecuzione siano ridotti drasticamente usando uno stream parallelo. Il vantaggio in termini di prestazioni è da calcolare rispetto alla stessa pipeline eseguita su uno stream non parallelo, e non alle stesse operazioni eseguite con cicli foreach o iteratori. Per esempio, supponiamo di voler contare quante volte una certa parola viene usata all'interno de "La Divina Commedia". Possiamo creare un programmino che compara le varie prestazioni per eseguire questo processo. Riportiamo qui sotto solo la parte relativa agli stream:

```
count = words.stream().filter(word ->
word.equals(wordToSearch)).count();
//...
count = words.parallelStream().filter(word ->
word.equals(wordToSearch)).count();
//...
```

Nella variabile `count` sarà memorizzato il numero di occorrenze della parola `wordToSearch`, trovate nel test. Ma il programma può effettuare la stessa

operazione anche eseguendo un ordinario ciclo foreach:

```
for (String word :  words) {
    if (word.equals(wordToSearch)) {
        count++;
    }
}
```

o usando un `Iterator`:

```
Iterator<String> iterator = words.iterator();
while (iterator.hasNext()) {
    if (iterator.next().equals(wordToSearch)) {
        count++;
   }
}
```

oppure un metodo `forEach()` direttamente sulla collezione:

```
int []countArray = {0};
words.forEach( word -> {if (word.equals(wordToSearch))
countArray[0]++;});
```

Tra un'esecuzione e un'altra registriamo i millisecondi impiegati per eseguire il processo. Il risultato finale (che cambia ad ogni esecuzione) è:

```
Tempo ciclo foreach    =  7  millisecondi   count  = 517
Tempo oggetto Iterator =  2  millisecondi   count  = 517
Tempo metodo forEach() =  41 millisecondi   count  = 517
Tempo stream ordinario =  10 millisecondi   count  = 517
Tempo stream parallelo =  51 millisecondi   count  = 517
```

Quindi come si può vedere gli stream non offrono in generale le prestazioni migliori. Questo dipende dal tipo di operazioni che vengono eseguite. Dando per scontato di avere sul proprio processore i necessari core per effettuare calcoli paralleli, l'effettivo vantaggio in termini di prestazioni si ottiene quando abbiamo situazioni in cui processando un certo elemento ci sia la possibilità di blocco. Tale blocco potrebbe essere determinato da un'eventuale attesa da parte del processo di input che arriva da fonti esterne rispetto alla nostra macchina, come una risorsa di rete o un database remoto (Input-Output e database sono argomenti rispettivamente del capitolo 18 e dell'appendice R). In tal caso uno stream parallelo potrebbe ridurre

drasticamente i tempi di processo totali. Infatti, se nel processare un certo elemento si verificasse questa attesa, solo il parallelismo ci permetterebbe di continuare la nostra esecuzione sugli altri elementi. In tutti gli altri casi invece il ciclo aspetterà che termini l'esecuzione del processo sull'elemento bloccante, sprecando tempo prezioso. Quello che bisogna evitare è usare il parallelismo senza che ce ne sia bisogno. Il fatto che basti invocare il metodo `parallelStream()` per implementare funzionalità di parallelismo, potrebbe risultare troppo invitante per alcuni programmatori poco esperti. In realtà si può incorrere facilmente in errori difficili da risolvere.

Riepilogo

La conoscenza approfondita del **framework Collections** è fondamentale per programmare in Java. L'interfaccia **`Collection`** è alla base della gerarchia del framework. Essa definisce diversi metodi fondamentali (`add()`, `addAll()`, `remove()`, `size()`, `iterator()`, `isEmpty()`, `contains()`, `toArray()`, etc.) che vengono implementati dalle sue sottoclassi. In particolare la **gerarchia del framework**, schematizzata nella figura 17.1, definisce come sottointerfacce di `Collection` le interfacce `Set`, `List`, `Queue` e `Deque`. `Set` viene estesa a sua volta da `SortedSet`, mentre un'altra gerarchia parte dall'interfaccia `Map`, che viene estesa da `SortedMap`. È possibile **iterare sulle collezioni** in vari modi. Il ciclo **foreach** dovrebbe essere utilizzato con collezioni di piccole dimensioni se non si devono fare operazioni impegnative sugli elementi. L'utilizzo di un **`Iterator`** invece rappresenta un modo efficiente per eseguire cicli sulle collezioni (di solito si utilizza il costrutto `while`). Il metodo **`forEach()`** ha il vantaggio di poter sfruttare gli oggetti `Stream` e le operazioni di aggregazione, definiti nella libreria Stream API. L'interfaccia **`List`** definisce metodi per i quali ogni sua implementazione rappresenta una collezione ordinata e indicizzata di elementi, in cui sono ammessi duplicati. Il metodo preferito per eseguire cicli sulle implementazioni di `List` è quello che usa il classico `for`, visto che le liste (come gli array) definiscono un indice per identificare ogni elemento. Le principali implementazioni di `List` sono `ArrayList` e `LinkedList`. **`ArrayList`** è probabilmente la collection più usata nella programmazione Java, e rappresenta una lista ridimensionabile non sincronizzata ad alte

prestazioni. Già dal nome si capisce come questa classe voglia rappresentare un array ridimensionabile. **`LinkedList`** invece è una **lista concatenata**, ovvero con ogni elemento che mantiene un reference verso l'elemento successivo e uno verso l'elemento precedente. Bisognerebbe preferire una `LinkedList` in luogo di `ArrayList` solo quando si devono aggiungere spesso elementi all'inizio della lista, oppure quando si vogliono eliminare elementi all'interno della lista durante le iterazioni. Queste operazioni richiedono infatti un **tempo costante** nelle `LinkedList` e un **tempo lineare** in un `ArrayList`. In compenso però, l'accesso posizionale in una `LinkedList` è lineare, mentre è costante in un'`ArrayList`. L'interfaccia **`Set`** rappresenta un **insieme matematico**, ovvero una collezione non ordinata di elementi in cui non sono ammessi duplicati. L'interfaccia **`SortedSet`** estende `Set` ed aggiunge la proprietà di essere ordinata. L'implementazione più famosa di `Set` è **`HashSet`**, mentre quella di `SortedSet` è **`TreeSet`** (che mantiene l'ordinamento ed è navigabile anche all'indietro). `HashSet` va preferita a `TreeSet` a meno che non si voglia iterare in maniera ordinata sugli elementi della collezione. Un'altra implementazione di `Set` è **`LinkedHashSet`** (che estende `HashSet`) che mantiene l'ordine degli elementi compatibile con il loro ordine di inserimento, come fanno le liste. Implementando anche l'interfaccia **`NavigableSet`** ha metodi supplementari per la navigazione come `lower()`, `floor()`, `ceiling()` e `higher()`, e `descendingIterator()`. Il framework contiene anche **metodi di utilità e algoritmi** già pronti per l'uso. Nella classe **`Collections`** vengono definiti tre tipi di metodi. I **metodi wrapper** come `unmodifiableList()` creano reference che rappresentano collezioni immutabili. Sono realmente immutabili se e solo se anche gli elementi della collezione sono immutabili e il reference originale della collezione è riassegnato al risultato del metodo wrapper. `Collections` definisce anche i metodi che abbiamo definito **sincronizzatori**, che consentono di ottenere copie thread-safe delle nostre collezioni. Anche in questo caso, se utilizziamo il reference originale, allora la collezione non sarà thread-safe. Come nel caso dei metodi wrapper bisogna quindi riassegnare il reference originale al risultato del metodo sincronizzatore. `Collections` definisce anche altri **metodi detti di convenienza** come `singleton()` e `nCopies()`. La classe **`Arrays`** contiene alcuni dei metodialgoritmi di `Collections`, ma relativi ad array, come `sort()`, e `parallelSort()`.

Definisce anche metodi molto usati come `asList()`, `toString()` che prende in input un array, e il metodo `copyOfRange()`. Java 9 introduce una nuova sintassi per la creazione di mappe, insiemi e liste immutabili. Infatti sono stati introdotti nelle interfacce `Map`, `Set` e `List`, dei **metodi statici di convenienza** che facilitano la creazione di collezioni immutabili. Dichiarando meno oggetti questa nuova sintassi permette di ottimizzare l'utilizzo della memoria. Si tratta di metodi factory di tipo `of()`, simili ai metodi `of()` che abbiamo visto nella libreria Date-Time API. Ci sono altre classi come **`Vector`**, **`Hashtable`** e **`Enumeration`**, considerate appartenenti al framework anche se secondarie rispetto alle altre, che sono argomento dell'appendice O. La libreria **Stream API** semplifica la programmazione Java soprattutto quando si usano le collezioni. Contemporaneamente offre supporto alla programmazione parallela. L'interfaccia **`Stream`** definita nel package **`java.util.stream`** rappresenta un flusso che ha come sorgente una collezione di dati, e su cui si possono eseguire operazioni complicate con uno sforzo di programmazione minimo, spesso basato su espressioni lambda e reference a metodi. Tipicamente uno stream viene creato a partire da una collection, tramite la chiamata al metodo `stream()`, ma ci sono diversi modi per ottenere oggetti `Stream`. È possibile istanziare uno `Stream` passando un array o direttamente dei valori al metodo statico `of()`, o tramite il metodo statico `generate()` della stessa classe `Stream`. Solitamente si itera sulle collezioni utilizzando un'oggetto `Stream`, che definisce anche un metodo **`forEach()`**, e da esso poi sfruttare le cosiddette **operazioni di aggregazione**, che per definizione sono **lazy**, ovvero sono chiamate solo all'ultimo momento, nel senso che saranno invocate solo quando sarà invocato il metodo terminale. Uno **stream** è una sequenza di elementi su cui possiamo iterare. L'iterazione è unica, e una volta terminata lo stream non è più utilizzabile. A differenza di una collection, uno stream non immagazzina elementi ma consente di operare su essi attraverso pipeline. Possiamo definire una **pipeline** come una sequenza di operazioni di aggregazione. Una pipeline è costituita da tre elementi: una **sorgente** (che potrebbe essere per esempio una collection o un array), zero o più **operazioni di aggregazione** (come il metodo `filter()`, i metodi `map()`, `mapToDouble()`, etc.) e un**'operazione terminale** (un metodo che restituisce un risultato che non è un oggetto `Stream`). Le **classi optional** possono immagazzinare valori di un certo tipo

onull, e permettono di evitare`NullPointerException`. Le classi `Optional` si possono creare con i metodi di tipo “of” come `of()` e `ofNullable()`. Mettono a disposizione diversi metodi molto utili come `orElse()`, `orElseGet()`, `orElseThrow()`, `get()`, `filter()`, `map()` e `flatMap()`. Le **reduction operation** (in italiano **operazioni di riduzione)** sono **operazioni terminali** che a partire dagli elementi di uno stream, ritornano un unico valore. Alcune implementazioni di operazioni di riduzione sono i metodi `average()`, `max()`, `min()`, `sum()` e `count()` che possono tutti essere considerati casi particolari del metodo **`reduce()`**. Il metodo **`collect()`** sfrutta i metodi statici della classe **`Collectors`** come `toList()`, `toSet()`, `toCollection()`, `joining()`, `groupingBy()` e i **metodi summarizing**, per aggregare risultati. Il metodo **parallelStreamO** restituisce un cosiddetto *stream parallelo*. Esso, per filtrare gli elementi della collezione, utilizza un algoritmo di tipo **fork/join.** Questo significa che divide la collezione in più parti e assegna l’operazione di filtraggio di ogni parte ad un thread diverso. Quando tutti i thread hanno terminato il loro processo, i risultati vengono ricongiunti per ottenere lo stesso risultato che si sarebbe ottenuto filtrando tutta la collezione interamente. Le prestazioni di uno stream parallelo risentono del lavoro extra fatto dall’algoritmo fork/join, e si fanno preferire rispetto a quelle di uno stream ordinario solo nei casi in cui c’è la possibilità di blocco durante l’algoritmo, per esempio causato dall’attesa dei risultati di sistema esterno come un database o un flusso di rete.

Esercizi e altro materiale didattico sono disponibili all’indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere l’utilità del framework Collections (unità 17.1)	☐	
	☐	

Conoscere le principali interfacce del framework Collections e i metodi che espongono (unità 17.2, 17.3, 17.4, 17.5, 17.6)		
Conoscere le principali implementazioni del framework Collections e i metodi che espongono (unità 17.2, 17.3, 17.4, 17.5, 17.6)	☐	
Saper utilizzare le classi d'utilità Arrays e Collections e i metodi che espongono (unità 17.7)	☐	
Comprendere l'utilità della libreria Stream API (unità 17.8)	☐	
Saper utilizzare stream, classi optional e pipeline (unità 17.8)	☐	
Saper utilizzare metodi di aggregazione, di riduzione e comprendere gli stream paralleli (unità 17.8)	☐	

Note:

18

Input-Output

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Aver compreso il pattern Decorator (unità 18.1, 18.2).
- ✔ Saper riconoscere nelle classi del package `java.io` i ruoli definiti nel pattern Decorator (unità 18.3).
- ✔ Capire le fondamentali gerarchie del package `java.io` (unità 18.3).
- ✔ Avere confidenza con i tipici problemi che si incontrano con l'input-output, come la serializzazione degli oggetti e la gestione dei file (unità 18.4).
- ✔ Avere confidenza con il nuovo modello per l'input output di Java 7 denominato NIO 2 (unità 18.5).

18.1 Introduzione all'input-output

Spesso le applicazioni hanno bisogno di utilizzare informazioni lette da **fonti esterne** o inviare informazioni a **destinazioni esterne.** Per informazioni intendiamo non solo stringhe, ma anche oggetti, immagini, suoni e così via. Per fonti o destinazioni esterne all'applicazione invece intendiamo file, dischi, reti, memorie o altri programmi. In questo capitolo vedremo come

Java consenta di gestire la lettura (**input**) da fonti esterne e la scrittura su destinazioni esterne **(output).** In particolare introdurremo il package `java.io`, croce e delizia dei programmatori Java e la libreria nota come **NIO 2**, che ha aggiunto altre funzionalità di alto livello per lavorare con i file in modo più semplice.

Il package `java.io` è molto vasto e anche abbastanza complesso. Conoscerne ogni singola classe è un'impresa ardua e soprattutto inutile. Per poter gestire l'input-output in Java conviene piuttosto capirne la filosofia che ne è alla base, regolata dal design pattern noto come Decorator (cfr. appendice D per la definizione di pattern). Non comprendere il pattern Decorator implicherà fare sempre fatica nel districarsi tra le classi di `java.io`. Al lettore quindi si raccomanda la massima concentrazione, visto che anche il pattern stesso è abbastanza complesso.

18.2 Pattern Decorator

È facile riconoscere nella gerarchia delle classi di `java.io` il modello di classi definito dal pattern Decorator. Si tratta di un pattern GoF strutturale (cfr. appendice D) decisamente complesso ma davvero potente. Nelle prossime righe viene proposto un esempio del pattern Decorator per poter dare un'idea al lettore della sua utilità. È però possibile (ma non consigliabile) passare direttamente a leggere la descrizione del package, se l'argomento pattern non interessa o lo si ritiene troppo complesso.

18.2.1 Descrizione del pattern

Il pattern **Decorator** consente al programmatore di implementare, tramite una particolare gerarchia di ruoli, una relazione tra classi che rappresenta un'alternativa dinamica alla statica ereditarietà. Implementando il pattern Decorator infatti, sarà possibile aggiungere responsabilità addizionali agli oggetti durante l'esecuzione, piuttosto che creare una sottoclasse per ogni nuova responsabilità. Sarà anche possibile ovviare a questo tipo di problema: supponiamo di voler aggiungere ad una certa classe, `ClasseBase`, delle responsabilità (chiamiamole `r1, r2, r3`, e `r4`) di pari dignità, ovvero ognuna indipendente dalle altre.

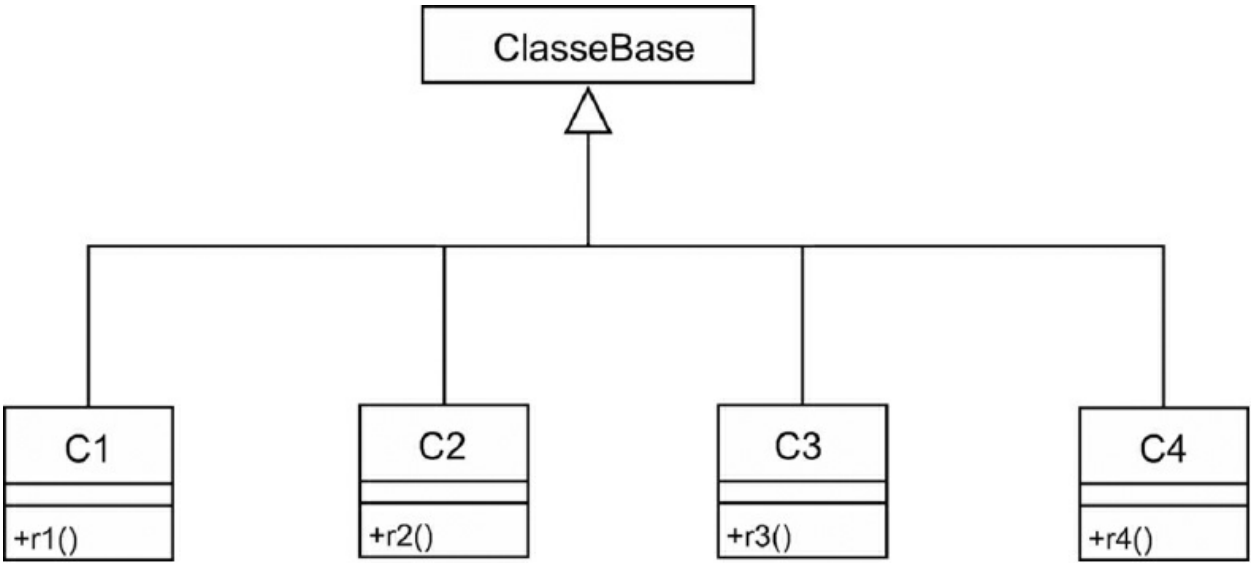


Figura 18.1 - Ereditarietà.

Potrebbe però servire anche creare classi che hanno più di una di queste responsabilità, per esempio `r1`, `r2` e `r4`. In questo caso particolare l'ereditarietà da sola non riesce a soddisfare i nostri bisogni. Partendo dalla gerarchia in figura 18.1 il lettore può provare a cercare una soluzione al problema. Probabilmente bisognerà creare un numero di classi pari a 2 elevato al numero delle varie responsabilità, nel nostro esempio 2 elevato alla quarta potenza, ovvero 16. Inoltre, causa l'impossibilità di implementare l'ereditarietà multipla con le classi, non sarà possibile ottenere un risultato accettabile dal punto di vista object oriented. Consigliamo al lettore però di non impegnarsi troppo per trovare una soluzione a questo problema, perché ne esiste una geniale già a disposizione di tutti: il pattern Decorator appunto. Nel pattern Decorator esistono i seguenti ruoli:

- `Component`: un'interfaccia (o classe astratta) che definisce una o più operazioni astratte da implementare nelle sottoclassi.
- `ConcreteComponent`: questo ruolo è interpretato da una o più classi non astratte che implementano `Component`.

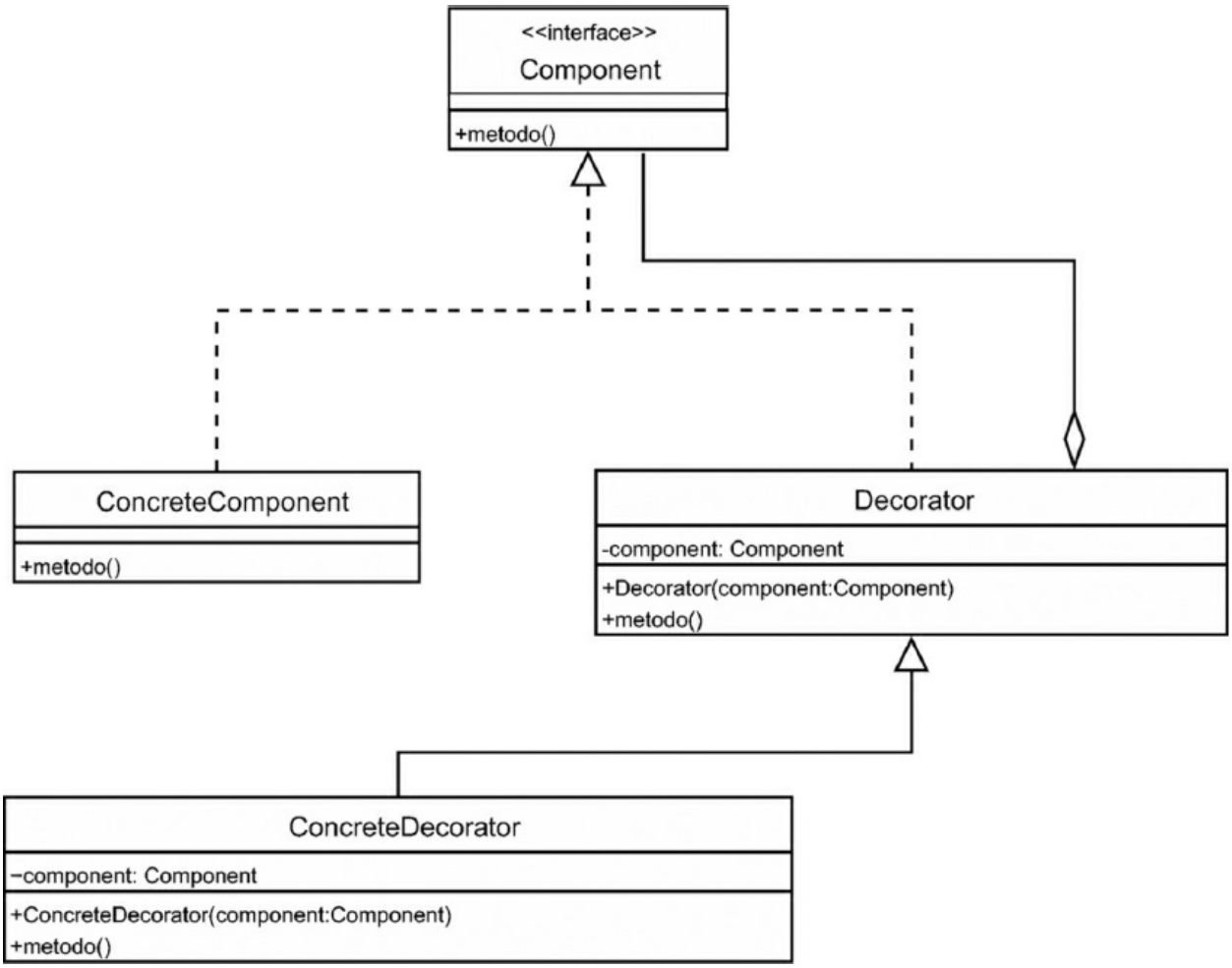


Figura 18.2 - Modello del pattern Decorator.

- `Decorator`: un'altra estensione di `Component` che potrebbe anche essere dichiarata astratta. Questa deve semplicemente obbligare le sue sottoclassi (`ConcreteDecorator`) non solo ad implementare `Component`, ma anche a referenziare un `Component` con un'aggregazione, ovvero una relazione tra classi che implica il contenimento (nell'esempio `Decorator` dichiara una variabile d'istanza di `Component`). La notazione di aggregazione in UML utilizza il simbolo del *diamante bianco* (figura 18.2, cfr. appendice L). Questo ruolo può essere considerato opzionale.
- `ConcreteDecorator`: implementazione di `Decorator`, che come abbiamo già asserito dovrà implementare `Component` e le sue operazioni e mantenere un riferimento verso un `Component`.

Tutto ciò viene riassunto con il diagramma in figura 18.2.

Facciamo un esempio per comprendere come funziona questo pattern.

18.2.2 Esempio

Come al solito è necessario calarsi in un contesto con un po' di fantasia e flessibilità. Supponiamo di voler realizzare un'applicazione grafica che consenta di aggiungere effetti speciali (come un effetto 3D e un effetto trasparente) alle nostre immagini. Per realizzare ciò occorre identificare la classe `Immagine` come `ConcreteComponent` e le classi `Effetto3DDecorator` e `TrasparenteDecorator` come `ConcreteDecorator`, come mostrato in figura 18.3.

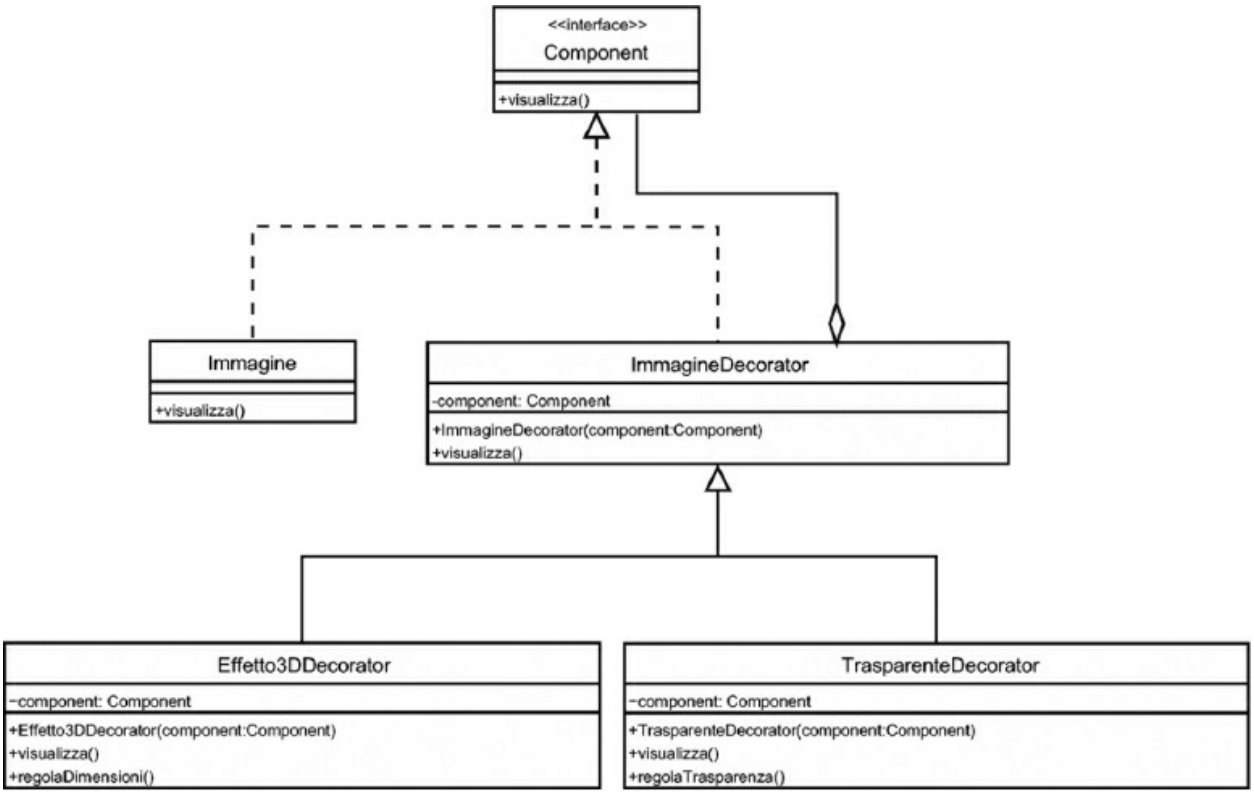


Figura 18.3 - Esempio di implementazione di Decorator.

A questo punto analizziamo il seguente codice:

```
1 Immagine monnaLisa = new Immagine();
2 monnaLisa.visualizza();
3 Effetto3DDecorator monnaLisa3D = new
Effetto3DDecorator(monnaLisa);
```

```
4 monnaLisa3D.visualizza();
5 TrasparenteDecorator monnaLisa3DTrasparente = new
    TrasparenteDecorator(monnaLisa3D);
6 monnaLisa3DTrasparente.visualizza();
```

Alle righe 1 e 2 viene istanziata e visualizzata l'immagine `monnaLisa`. Alla riga 3 entra in scena il primo decoratore che viene istanziato aggregandogli l'oggetto `monnaLisa` appena istanziato. Questo è il punto più complesso e ha bisogno di essere ben analizzato. Per prima cosa notiamo che ogni decoratore ha un unico costruttore che prende in input obbligatoriamente (concetto di aggregazione) un `Component`. Ovvero non può esistere un oggetto `Decorator` senza che aggreghi un `Component` (in questo caso `Immagine` ha ruolo di `ConcreteComponent`). Di fatto un effetto speciale non può esistere senza l'oggetto da decorare. È questo il punto chiave per comprendere il pattern. Alla riga 4 è invocato il metodo `visualizza()` direttamente sull'oggetto `Effetto3DDecorator (monnaLisa3D)` che contiene l'immagine. Notiamo come l'oggetto `monnaLisa3D` (di tipo `Effetto3DDecorator`) se da un lato rappresenta solo una decorazione dell'immagine, in questo pattern diventa proprio l'oggetto immagine decorato. Ecco perché abbiamo preferito chiamarlo `monnaLisa3D` nonostante sia di tipo `Effetto3DDecorator`. Ovviamente, sarebbe più naturale che un oggetto di tipo `Immagine` (o una sua sottoclasse) rimanesse l'oggetto contenitore, ma bisogna abituarsi a questa nuova idea per ottenere i benefici di questo pattern.

Si noti inoltre che il metodo `visualizza()` di un `Decorator` farà sicuramente uso anche del metodo `visualizza()` dell'`Immagine`. Per esempio il metodo `visualizza()` della classe `Effetto3DDecorator` potrebbe essere codificato più o meno nel modo seguente:

```
public void visualizza() {
    //codice per generare l'effetto 3D...
    component.visualizza();
}
```

Il reference `component` punta all'oggetto `monnaLisa` di tipo `Immagine`.

Alla riga 5 si ripete un'altra decorazione, ma stavolta il decoratore `(monnaLisa3DTrasparente` di tipo `TrasparenteDecorator)` invece di decorare un oggetto di tipo `Immagine`, agisce su un altro decoratore

(`monnaLisa3D` di tipo `Effetto3DDecorator`) che ha già operato su un oggetto di tipo `Immagine` (`monnaLisa`). Ciò è possibile grazie al fatto che ogni `Decorator` deve decorare per forza un `Component`, e `Component` è implementato anche dai `Decorator`. Alla riga 6 viene invocato il metodo `visualizza()` direttamente sull'oggetto `TrasparenteDecorator` (`monnaLisa3DTrasparente`) che contiene il decoratore contenente a sua volta l'immagine. Anche in questo caso il metodo `visualizza()` di `TrasparenteDecorator` farà sicuramente uso anche del metodo `visualizza()` del `component` che aggrega. Stavolta però il `component` aggregato non è una semplice immagine ma un oggetto di tipo `Effetto3DDecorator` (`monnaLisa3D`) che a sua volta aggregava l'oggetto `monnaLisa` di tipo `Immagine`. Concludendo, il pattern è sicuramente complesso e questo non gioca a suo favore, ma è incredibilmente potente. Basti pensare che, facendo riferimento all'esempio appena presentato, esistono almeno i seguenti vantaggi:

1. il numero delle classi da creare è molto minore che in qualsiasi soluzione basata sulla ereditarietà. Per esempio non deve esistere la classe `Immagine3DTrasparente`;
2. se nascono nuovi effetti speciali, basterà aggiungerli come decoratori senza che tutto il resto del codice ne risenta;
3. durante l'esecuzione si può creare qualsiasi combinazione basata sui decoratori senza sforzi eccessivi.

18.3 Caratteristiche del package `java.io`

L'implementazione del pattern Decorator per la gestione dell'input-output in Java si è sicuramente rivelata la soluzione ideale. Infatti, come già asserito nell'introduzione, tale package deve mettere a disposizione dell'utente classi per realizzare un qualsiasi tipo di lettura (input) e un qualsiasi tipo di scrittura (output). Le fonti di lettura e le destinazioni di scrittura sono molte e in futuro potrebbero nascerne di nuove. Il pattern Decorator permette quindi di realizzare qualsiasi tipo di comunicazione con fonti di destinazioni esterne usando un limitato numero di classi. Nonostante ciò, il numero di classi del package `java.io` rimane comunque alto. Fortunatamente non bisogna

conoscerle tutte; la documentazione ufficiale serve proprio a questo e, conoscendo il pattern Decorator, potremo riconoscere i ruoli di ogni classe.

Partiamo dal concetto fondamentale che è alla base del discorso: lo **stream** (in italiano **flusso, e da non confondere con il concetto di stream visto nel precedente capitolo).** Per prelevare informazioni da una fonte esterna (un file, una rete, etc.), un programma deve aprire uno stream su di essa e leggerne le informazioni in modo sequenziale. La figura 18.4 mostra graficamente l'idea.

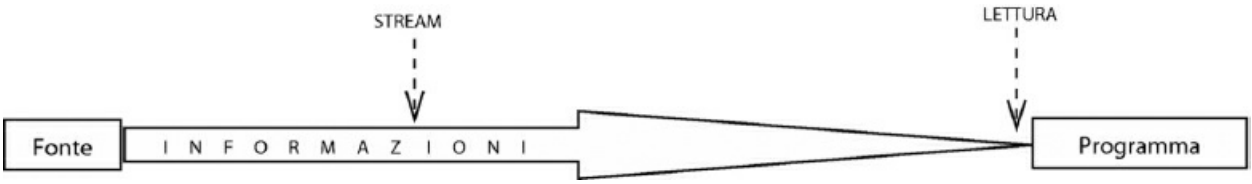


Figura 18.4 - Rappresentazione grafica di un input.

Allo stesso modo un programma può inviare informazioni ad una destinazione esterna aprendo uno stream su essa e scrivendole sequenzialmente, come mostrato in figura 18.5.

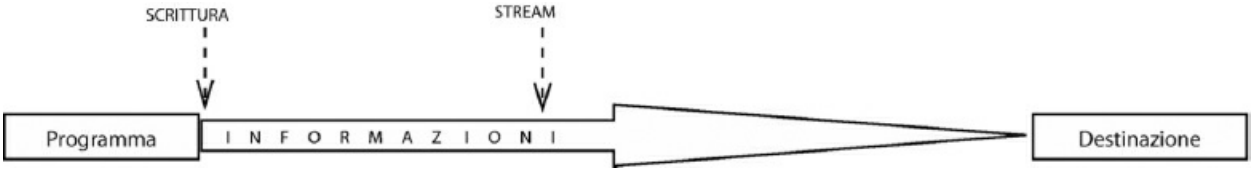


Figura 18.5 - Rappresentazione grafica di un output.

In altre parole, aprire uno stream, da una fonte o verso una destinazione, significa aprire verso questi punti terminali un canale di comunicazione dove far passare le informazioni. Con `java.io` non importa di che tipo siano le informazioni, né con che fonti o destinazioni abbiamo a che fare. Gli algoritmi per scrivere o leggere, infatti, sono sostanzialmente sempre gli stessi.

Per l'input è necessario:

1. aprire lo stream;
2. leggere tutte le informazioni dallo stream fino a quando non terminano;

3. chiudere lo stream.

E per l'output bisogna:

1. aprire lo stream;
2. scrivere tutte le informazioni tramite lo stream fino a quando non terminano;
3. chiudere lo stream.

Il package `java.io` contiene una collezione di classi che supportano tali algoritmi per leggere e scrivere. Le classi di tipo stream sono divise in due gerarchie separate (anche se simili) in base al tipo di informazione che devono trasportare (byte o caratteri).

18.3.1 I Character Stream

`Reader` e `Writer` sono le due superclassi astratte per i *character stream (flussi di caratteri)* in `java.io`. Queste due classi hanno la caratteristica di obbligare le sottoclassi a leggere e scrivere dividendo i dati in pezzi di 16 bit ognuno, quindi compatibili con il tipo `char` di Java. Le sottoclassi di `Reader` e `Writer` hanno i ruoli del pattern Decorator. Il ruolo di `Component` è interpretato proprio da `Reader` (e `Writer`). Le sottoclassi di `Reader` (e `Writer`) implementano stream speciali. Alcune di queste hanno il ruolo di `ConcreteComponent` (dovremmo dire `ConcreteReader` e `ConcreteWriter`) e da sole possono attaccarsi ad una fonte (o ad una destinazione) e leggere (o scrivere) subito, quantomeno con un algoritmo sequenziale implementato nei metodi `read()` (o `write()`) ereditati da `Reader` (o `Writer`). Questi stream sono anche detti **node stream** (in italiano **flussi nodo**).

Altre sottoclassi di `Reader` (e `Writer`) invece interpretano il ruolo di `ConcreteDecorator`. Tali stream sono anche detti **processing stream** e senza aggregazioni a `ConcreteReader` (o `ConcreteWriter`) non si possono nemmeno istanziare. Il più delle volte lo scopo dei decoratori di stream è quello di migliorare le prestazioni o facilitare la lettura (o la scrittura) delle informazioni negli stream, fornendo metodi adeguati allo scopo, ed evitando così cicli noiosi e poco eleganti. Nelle figure 18.6 e 18.7 sono mostrate le gerarchie delle classi principali per quanto riguarda rispettivamente `Reader` e

`Writer`, con i decoratori colorati in grigio. I programmi dovrebbero utilizzare i character stream (con i reader e i writer) per le informazioni di tipo testuale. Infatti questi permettono di leggere qualsiasi tipo di carattere Unicode.

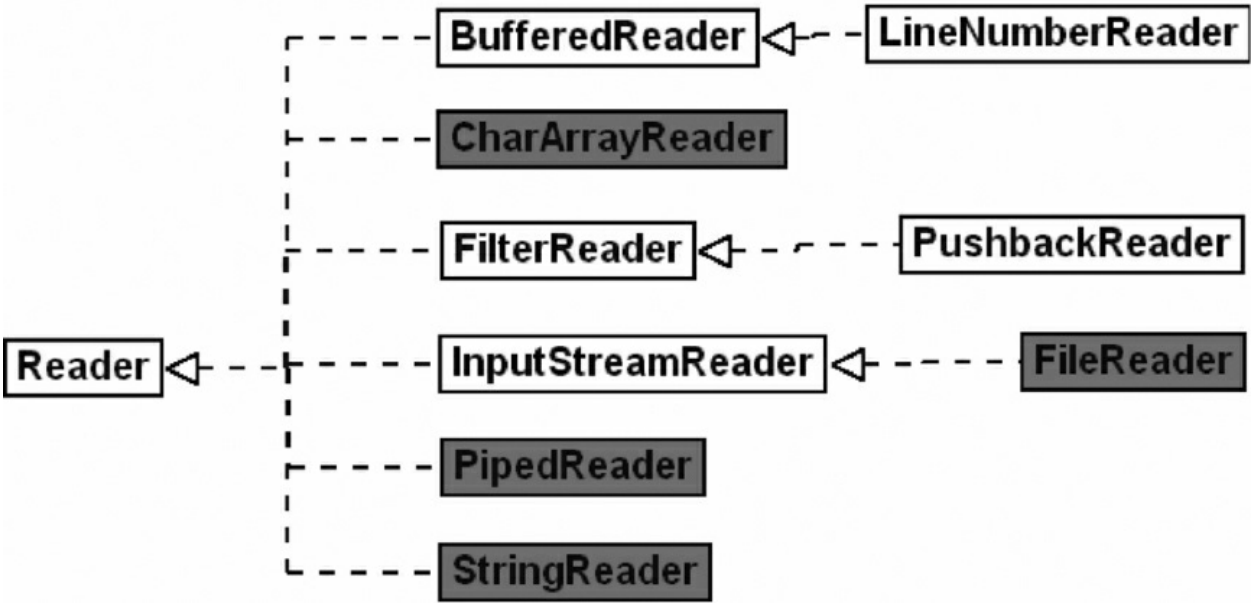


Figura 18.6 - Gerarchia dei Reader.

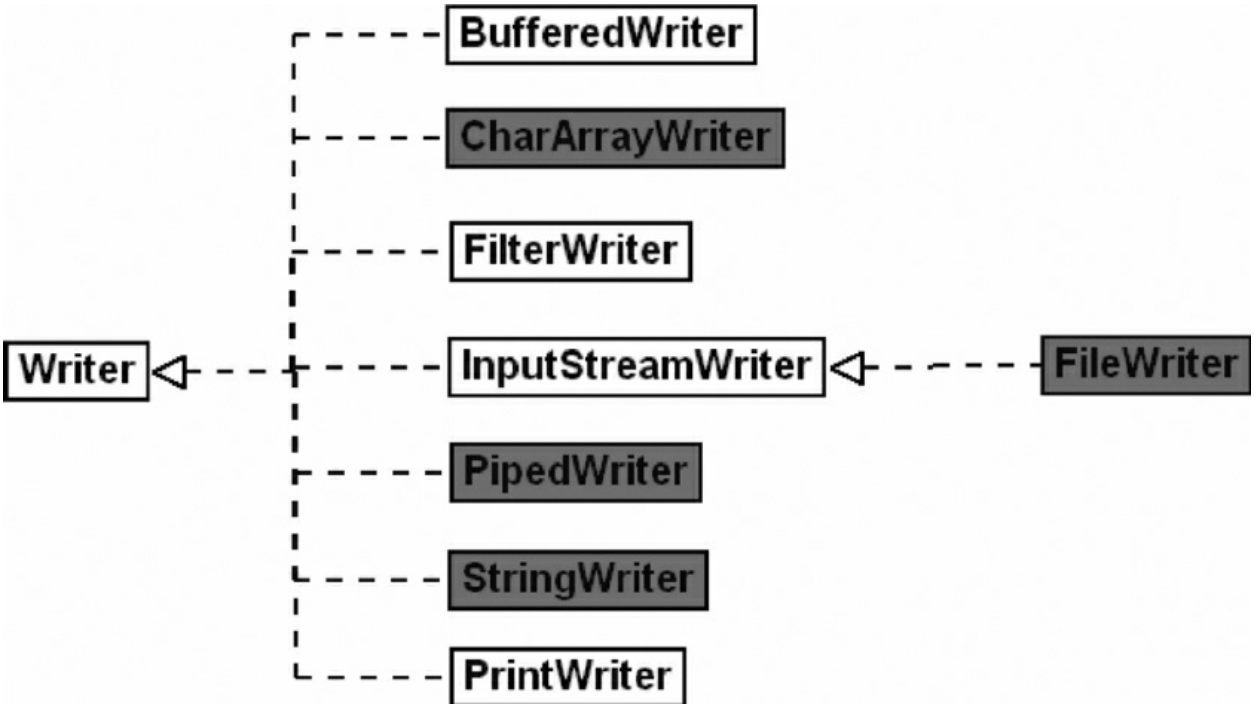


Figura 18.7 - Gerarchia dei Writer.

18.3.2 I Byte Stream

Nel package `java.io` esistono gerarchie di classi parallele ai reader e ai writer che però sono destinate alla lettura e alla scrittura di informazioni non testuali (per esempio file binari come immagini o suoni). Queste classi infatti leggono (e scrivono) dividendo i dati in pezzi di 8 bit ognuno. Quanto detto per `Reader` e `Writer` vale anche per le classi `InputStream` e `OutputStream`. Le figure 18.8 e 18.9 mostrano le gerarchie delle classi principali per quanto riguarda rispettivamente `InputStream` e `OutputStream`, con i decoratori colorati in grigio.

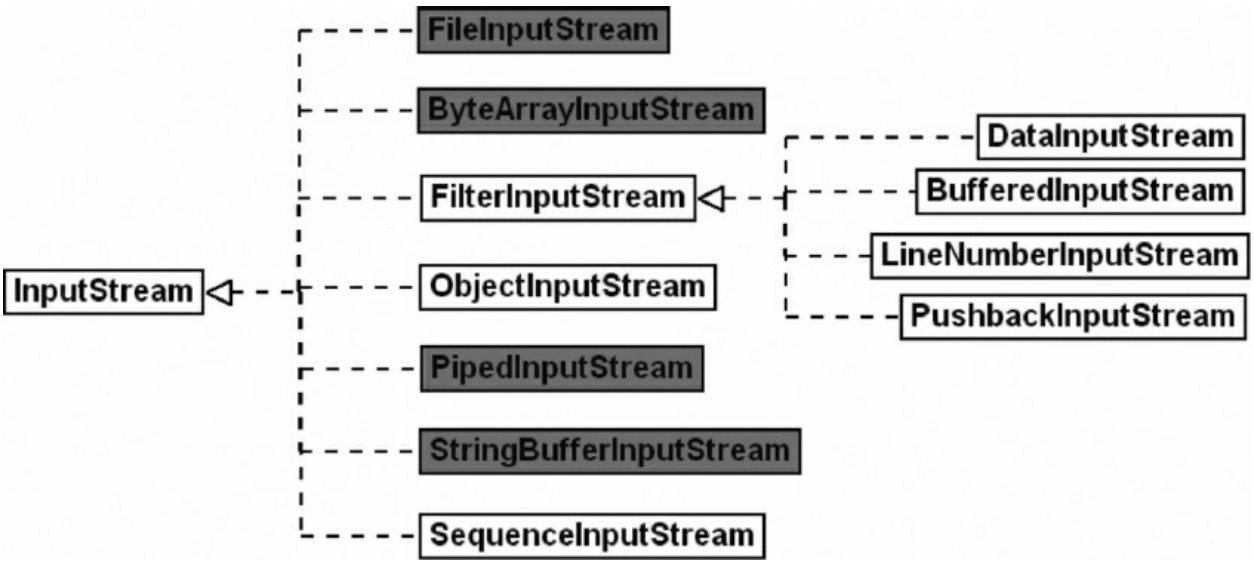


Figura 18.8 - Gerarchia degli InputStream.

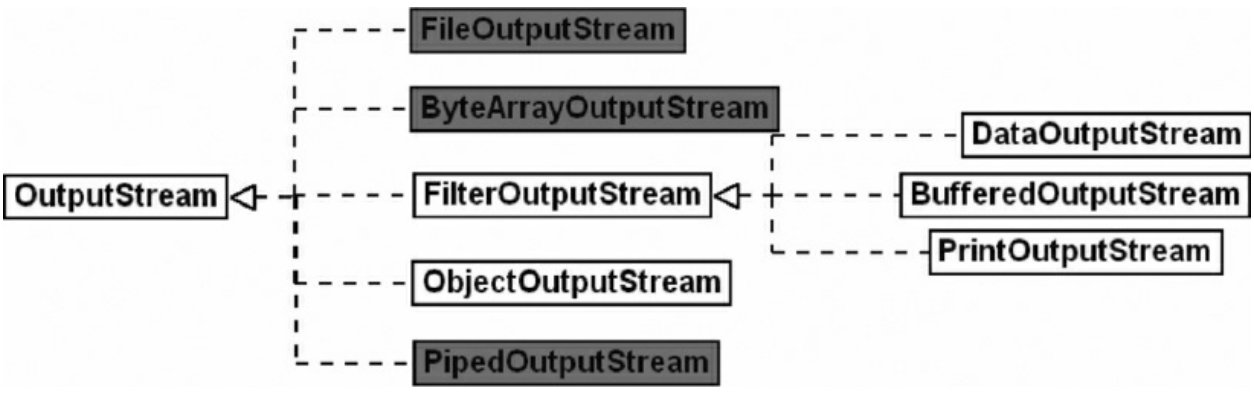


Figura 18.9 - Gerarchia degli OuputStream.

Due di queste classi, `ObjectInputStream` e `ObjectOutputStream`, sono utilizzate per la cosiddetta serializzazione di oggetti, di cui vedremo un

esempio nel paragrafo 18.4.3. Per esempio, con le seguenti righe di codice creiamo la copia di un file:

```
InputStream inputStream = new
FileInputStream("C:\primoFile");
OutputStream outputStream = new
FileOutputStream("C:\secondoFile");
byte[] byteBuffer = new byte[1024];
int bytesRead = 0;
while ((bytesRead = inputStream.read(byteBuffer)) >= 0) {
    outputStream.write(byteBuffer, 0, bytesRead);
}
outputStream.close();
inputStream.close();
```

Dapprima apriamo il node stream `FileInputStream` che attacchiamo alla fonte, ovvero una stringa contenente il nome di un file (`C:\primoFile`). In questo modo abbiamo anche già aperto un flusso di lettura verso il file. Poi abbiamo aperto un flusso di scrittura verso un altro file (`C:\secondoFile`) istanziando un `FileOutputStream` in cui copieremo il primo file. Poi dichiariamo un array di 1024 byte in cui scriveremo i byte letti dal primo file, per poi scriverli all'interno del secondo file. La variabile locale `bytesRead`, avrà il compito di conservare il numero di byte letti, ed è inizialmente impostata a `0`. Il ciclo `while` seguente, esegue l'algoritmo che copia il primo file nel secondo file. Nelle parentesi tonde dove si specifica la condizione del ciclo `while`, troviamo una sintassi forse mai incontrata sinora ma del tutto legale. Infatti tra delle paretesi tonde innestate troviamo anche l'assegnazione alla variabile locale `bytesRead`. Il metodo `read()` chiamato sull'oggetto `inputStream` di tipo `FileInputStream`, prende in input l'array di byte `byteBuffer`. Questo metodo legge i byte dal file sorgente (`C:\primoFile`) fino a riempire l'array di byte `byteBuffer` (quindi può leggere al massimo 1024 byte alla volta). Il metodo restituisce il numero di byte letti e li immagazzina nella variabile `bytesRead`. Quindi, se per esempio il primo file è grande 1025 byte, questo ciclo eseguirà due iterazioni utili: nella prima la variabile `bytesRead` avrà valore `1024`, nella seconda avrà valore `1`. Nella terza iterazione leggerà il valore di default `-1`, che viene ritornato se non ci sono altri byte da leggere. Questo provocherà la fine del ciclo. Ad ogni iterazione, il corpo del costrutto `while` scrive nell'oggetto `outputStream` di tipo

`FileOutputStream` (e di conseguenza nel secondo file) tramite il metodo `write()`, quanto appena letto dal primo file. Infatti al metodo `write()` vengono passati l'array di byte `byteBuffer`, l'indice iniziale dell'array da cui iniziare a prelevare i byte da scrivere (`0`), e il numero di byte da scrivere (`bytesRead`). Infine chiudiamo con il metodo `close()` entrambi gli stream.

18.3.3 Le interfacce alla base della gerarchia

`Reader` e `InputStream` definiscono praticamente gli stessi metodi ma per differenti tipi di dati. Per esempio, `Reader` contiene i seguenti metodi per leggere caratteri ed array di caratteri:

- `int read() throws IOException`
- `int read(char cbuf[]) throws IOException`
- `int read(char cbuf[], int offset, int length)`
- `throws IOException`

`InputStream` definisce gli stessi metodi, ma per leggere byte ed array di byte:

- `int read() throws IOException`
- `int read(byte cbuf[]) throws IOException`
- `int read(byte cbuf[], int offset, int length)`
- `throws IOException`

Inoltre `InputStream` fornisce il seguente metodo d'utilità:

- `int available() throws IOException` restituisce il numero di byte che possono essere letti all'interno dello stream.

Mentre sia `Reader` che `InputStream` forniscono i seguenti metodi d'utilità:

- `void close() throws IOException` rilascia le risorse di sistema associate allo stream (ma non provoca una deallocazione immediata);

- `long skip(long nbytes) throws IOException` prova a leggere e a scartare un certo numero di byte definito dal numero `nbytes`, e ritorna il numero di byte scartati.

Anche `Writer` e `OutputStream` sono da considerarsi classi parallele. `Writer` definisce i seguenti metodi per scrivere caratteri ed array di caratteri:

- `int write(int c) throws IOException`
- `int write(char cbuf[]) throws IOException`
- `int write(char cbuf[], int offset, int length)`
- `throws IOException`

`OutputStream` invece definisce gli stessi metodi ma per i byte:

- `int write(int c) throws IOException`
- `int write(byte cbuf[]) throws IOException`
- `int write(byte cbuf[], int offset, int length)`
- `throws IOException`

Quest'ultimo metodo l'abbiamo già incontrato nell'esempio del paragrafo precedente. Si noti come tutti questi metodi dichiarino nella clausola `throws` la checked exception `IOException`. Quindi ogni volta che li usiamo dobbiamo o gestire o rilanciare tale eccezione.

18.3.4 Checked exception e chiusura degli stream

Tutti gli stream (reader, writer, input stream e output stream) sono automaticamente aperti quando creati. Ogni stream si dovrebbe chiudere esplicitamente chiamando il metodo `close()`. Il garbage collector infatti, non può chiudere uno stream se questo è ancora aperto, e di conseguenza non potrà neanche deallocarne la memoria.

Conviene quindi chiudere gli stream esplicitamente ogni volta che è possibile per migliorare le prestazioni dell'applicazione e per non creare **memory leak** (la mancata deallocazione della memoria non più utilizzata con

conseguente spreco di memoria). Una buona tecnica è chiudere lo stream nella clausola `finally` di un blocco `try-catch`. Per esempio consideriamo il seguente esempio che copia un file, ma in una situazione in cui c'è una classe con un metodo e il blocco `try-finally` che garantisce la chiusura degli stream:

```
import java.io.*;

public class CloseResources {
    public static void copyFile(String source, String
destination)
      throws FileNotFoundException, IOException {
        InputStream inputStream = new
FileInputStream(source);
        OutputStream outputStream = new
FileOutputStream(destination);
        try {
            byte[] byteBuffer = new byte[1024];
            int bytesRead = 0;
            while ((bytesRead =
inputStream.read(byteBuffer)) >= 0) {
                outputStream.write(byteBuffer, 0,
bytesRead);
            }
       } finally {
            outputStream.close();
            inputStream.close();
       }
    }
}
```

Si noti che il metodo `copyFile()` dichiara nella clausola `throws` la `FileNotFoundException` e la `IOException`. Infatti se provassimo a commentare la clausola, otterremo i seguenti errori in compilazione:

```
CloseResources.java:6: error: unreported exception
  FileNotFoundException; must be caught or declared to be
thrown
        InputStream inputStream = new FileInputStream(source);
                                  ^
CloseResources.java:7: error: unreported exception
  FileNotFoundException; must be caught or declared to be
thrown
```

```
        OutputStream outputStream = new
FileOutputStream(destination);
                                       ^
CloseResources.java:15: error: unreported exception
IOException;
  must be caught or declared to be thrown
           outputStream.close();
                               ^
CloseResources.java:16: error: unreported exception
IOException;
  must be caught or declared to be thrown
           inputStream.close();
                              ^
CloseResources.java:11: error: unreported exception
IOException;
  must be caught or declared to be thrown
           while ((bytesRead = inputStream.read(byteBuffer))
>= 0)
                                                    ^
CloseResources.java:12: error: unreported exception
IOException;
  must be caught or declared to be thrown
           outputStream.write(byteBuffer, 0, bytesRead);
                               ^
6 errors
```

Questo perché come abbiamo visto, i metodi `write()`, `read()` e `close()` utilizzati rilanciano la checked exception `IOException`. Invece i costruttori di `FileInputStream` e `FileOutputStream` rilanciano la checked exception `FileNotFoundException`. In particolare il costruttore di `FileInputStream` lancerebbe tale eccezione se il nome del file non coincidesse con un nome di un file esistente, mentre il costruttore di `FileOutputStream` la solleverebbe se non riuscisse a creare il file di destinazione (per esempio se l'utente che ha eseguito il programma non ha i privilegi di sistema per creare file, o se il nome del file coincide con quello di una directory e così via). Questo implica che se volessimo gestire tutte le eccezioni bisognerebbe aggiungere le giuste clausole `catch`. La gestione più noiosa, sarebbe quella che circonderà con dei blocchi `try-catch` anche i metodi `close()` all'interno del `finally`:

```
 finally {
      try {
          outputStream.close();
```

```
        } catch(IOException) {
            // gestione dell'eccezione
        }
        try {
            inputStream.close();
        } catch(IOException) {
            // gestione dell'eccezione
        }
    }
```

Purtroppo uno dei punti deboli di questa libreria, è proprio l'utilizzo costante delle checked exception che ci obbliga a gestire tutte le eccezioni. Se da un lato questo rende l'applicazione più robusta, dall'altra rende Java un linguaggio troppo verboso.

Per ridurre la verbosità e per lo meno evitare di chiamare il metodo `close()` ogni volta esplicitamente ci viene in aiuto il costrutto **try with resources**, già introdotto nel paragrafo 9.4. Molte classi della libreria standard implementano l'interfaccia `Closeable` o `AutoCloseable`. Questo rende tali classi eleggibili come parametri per il costrutto try with resources. Tra queste classi ci sono tantissime classi ed interfacce del package `java.io` come `InputStream` e `OutputStream`. Segue il precedente esempio modificato per supportare tale costrutto:

```
  public static void copyFile(String source, String
  destination) throws
    FileNotFoundException, IOException {
      try (InputStream inputStream = new
  FileInputStream(source);
           OutputStream outputStream = new
  FileOutputStream(destination)) {
          byte[] byteBuffer = new byte[1024];
          int bytesRead = 0;
          while ((bytesRead = inputStream.read(byteBuffer)) >=
  0) {
              outputStream.write(byteBuffer, 0, bytesRead);
          }
     }
  }
```

Si può notare una minore verbosità del codice in quanto i flussi non sono stati

chiusi esplicitamente, ma sono chiusi automaticamente dal costrutto try with resources. Questo è un grande aiuto visto che uno degli errori più frequenti nella programmazione Java è proprio quello di dimenticare di chiudere le risorse causando un memory leak.

18.4 Input e output classici

In questo paragrafo saranno presentati alcuni esempi di classiche situazioni di utilizzo dell'input-output.

18.4.1 Lettura di input da tastiera

Il seguente codice definisce un programma che riesce a leggere ciò che l'utente scrive sulla tastiera e, dopo la pressione del tasto Invio, ristampa quanto letto. Per terminare il programma è necessario digitare "fine" e premere Invio.

> **Se si esegue quest'applicazione con EJE, per potere iniziare a digitare con la tastiera bisognerà posizionarsi sull'area di output dell'editor.**

```
import java.io.*;

public class KeyboardInput {
    public static void main(String args[]) throws
IOException {
        String stringa = null;
        System.out.println("Digita qualcosa e premi
invio...\n"
                    + "Per terminare il programma digitare
\"fine\"");
        try (InputStreamReader ir = new
InputStreamReader(System.in);
          BufferedReader in = new BufferedReader(ir)) {
            stringa = in.readLine();
            while ( stringa != null ) {
                if (stringa.equals("fine")) {
                    System.out.println("Programma
terminato");
```

```
                    break;
                }
                System.out.println("Hai scritto: " +
stringa);
                stringa = in.readLine();
            }
        }
    }
}
```

Nel blocco `try` un `InputStreamReader` viene creato ed agganciato all'oggetto `System.in`, che rappresenta l'input di default del nostro sistema (molto probabilmente la tastiera).

> **La classe `InputStreamReader` consente di trasformare una fonte dati di tipo `byte` in una fonte dati di tipo `char`.**

L'oggetto `isr` da solo ci avrebbe già permesso di leggere con uno dei metodi `read()` il contenuto dallo stream proveniente dalla tastiera, ma sarebbe stato molto più complesso realizzare il nostro obiettivo. Infatti avremmo dovuto concretizzare il seguente algoritmo:

1. leggere ogni byte con un ciclo;
2. per ogni carattere letto con il metodo `read()`;
   - **I.** controllare se il carattere coincide con "`\n`" (ovvero il tasto Invio);
   - **II.** se è verificato il passo I, stampare i caratteri precedenti e svuotare lo stream. Se i caratteri precedenti formano la parola "fine", terminare l'applicazione. Se la condizione I non è verificata, leggere il prossimo carattere.

Si noti come invece, decorando con un `BufferedReader` l'oggetto `isr`, la situazione si sia notevolmente semplificata. Il `BufferedReader` infatti mette a disposizione del programmatore diversi metodi, tra cui il metodo `readLine()`, che restituisce il contenuto dello stream sotto forma di stringa fino al tasto Invio.

Quanta strada abbiamo dovuto fare con Java per leggere finalmente un carattere da tastiera! L'input-output di Java è effettivamente un argomento complesso, ma come constateremo presto la complessità è costante per qualsiasi operazione di input-output. Per esempio, il metodo `println()` viene utilizzato sia per stampare sulla prompt DOS sia per stampare contenuto dinamico in una risposta HTTP o in un flusso viaggiante in rete (cfr. appendice P on line dedicata al networking).

`BufferedReader` definisce anche il metodo `lines()` che restituisce uno `java.util.stream.Stream` processato in maniera *lazy*. Per esempio potremmo leggere da una fonte tutte le righe per trovare la prima riga che contiene la parola "claudio":

```
Stream<String> lines = reader.lines();
Optional<String> claudio =
  lines.filter(s -> s.contains("claudio")).findFirst();
claudio.ifPresent(System.out::println);
```

Abbiamo già parlato del metodo `printf()` nel paragrafo 14.1.4 dedicato alle formattazioni di output. Si trova nella classe `PrintStream` (la classe di `System.out`) e basa la sua implementazione sul metodo `format()` della classe `java.util.Formatter`.
In Java è ora possibile scrivere il seguente codice:

```
boolean b = true;
System.out.printf("Ecco un valore booleano: %b, ora si
formatta come in C!", b);
```

ottenendo il seguente output:

```
Ecco un valore booleano: true, ora si formatta come in C!
```

Inoltre la classe `Scanner` del package `java.util` consente di semplificare la lettura di sorgenti di input, siano esse stringhe, tipi primitivi, file o altro. Per esempio, è possibile leggere da tastiera tramite `Scanner` con il seguente codice:

```
try (Scanner sc = new Scanner(System.in)) {
    String testoDigitato = "";
    while (sc.hasNext()) {
        testoDigitato = sc.nextLine();
```

```
            System.out.println(testoDigitato);
        }
    }
```

> **Si noti che anche `Scanner`, implementando `Closeable`, è stato gestito con il costrutto "try with resources".**

`Scanner` apre un flusso di input verso la tastiera (`System.in`) e leggerà i caratteri immessi da tastiera nel momento in cui sarà premuto il tasto Invio. I metodi `hasNext()` e `nextLine()` ricordano i metodi di `StringTokenizer` o `Iterator`. Il primo restituisce il valore `true` se c'è qualcosa da leggere (ovvero sarà eseguito quando l'utente premerà il tasto Invio), mentre il secondo leggerà la stringa digitata. `Scanner` può utilizzare anche le espressioni regolari (cfr. paragrafo 14.1.5) per realizzare complesse operazioni di riconoscimento di testo.

18.4.2 Gestione dei file

Nel package `java.io` esiste la classe `File` che astrae il concetto di file generico.

Anche una directory è un file: un file che contiene altri file.

Seguono i dettagli dei metodi più interessanti di questa classe:

- `boolean exists()` restituisce `true` se l'oggetto file coincide con un file esistente sul file system.
- `String getPath()` ritorna il path assoluto del file, ovvero il percorso completo compreso di nome del file.
- `String getName()` restituisce il nome del file (o della directory).
- `String getParent()` restituisce il nome della directory che contiene il file.
- `boolean isDirectory()` restituisce `true` se l'oggetto file coincide con una directory esistente sul file system.

- `boolean isFile()` restituisce `true` se l'oggetto file coincide con un file esistente sul file system.
- `String[] list()` ritorna una array di stringhe contenente i nomi dei file presenti nella directory su cui viene chiamato il metodo. Se questo metodo è invocato su un file che non è una directory, restituisce `null`.
- `boolean delete()` tenta di eliminare il file corrente.
- `long length()` restituisce la lunghezza del file in byte.
- `boolean mkdir()` tenta di creare una directory il cui path è descritto dall'oggetto `File` corrente.
- `boolean renameTo(File newName)` tenta di rinominare il file corrente. Tale metodo restituisce `true` se ha successo.
- `boolean canRead()` restituisce `true` se il file o la directory sono leggibili dall'utente corrente (ovvero se ha permesso in lettura).
- `boolean canWrite()` restituisce `true` se il file o la directory sono modificabili.
- `boolean createNewFile()` crea un nuovo file vuoto come descritto dall'oggetto corrente se tale file non esiste già. Restituisce `true` se e solo se tale file viene creato.

I costruttori della classe `File` sono i seguenti:

- `File(String pathname)`
- `File(String dir, String subpath)`
- `File(File dir, String subpath)`

Di seguito qualche esempio:

```
File dir = new File("/usr", "local");
File file = new File(dir, "Abc.java");
File dir2 = new File("C:\\directory");
File file2 = new File(dir2, "Abc.java");
```

Si noti come, con le prime due righe, abbiamo istanziato una directory e un

file su un sistema Unix, mentre con le ultime due righe abbiamo eseguito le stesse operazioni su un sistema Windows. Si noti anche che abbiamo utilizzato il separatore per il path dei file per i vari sistemi operativi in maniera dipendente, ma è anche possibile utilizzare come separatore `/` pure su sistemi Windows. Se vogliamo essere più precisi (ma a costo di maggiore verbosità) possiamo utilizzare anche la variabile statica `pathSeparator` della classe `File`. Questa varrà `\` su sistemi Windows e `/` su sistemi Unix. Per esempio, si può creare un file nel seguente modo:

```
File file = new File(".." + File.pathSeparator +
"Abc.java");
```

Istanziare un file non significa però crearlo fisicamente sul file system; per farlo è necessario utilizzare gli output stream (o eventualmente writer nel caso di file di testo). Il seguente codice consente di creare una copia di backup di un file specificato da riga di comando:

```
import java.io.*;

public class BackupFile {
    public static void main(String[] args) throws
IOException {
        if (args.length == 0) {
            System.out.println("Specificare nome del
file!");
            System.exit(0);
        }
        File inputFile = new File(args[0]);
        File outputFile = new File(args[0] + ".backup");
        try (FileInputStream fis = new
FileInputStream(inputFile);
             FileOutputStream fos = new
FileOutputStream(outputFile)) {
            int b = 0;
            while ((b = fis.read()) != -1) {
                fos.write(b);
            }
            System.out.println("Eseguito backup in " +
args[0] + ".backup!");
        }
    }
}
```

Il codice precedente ha bisogno che sia specificato il nome del file di cui eseguire la copia, come argomento da riga di comando, altrimenti termina stampando un messaggio esplicativo. Il codice è abbastanza semplice. Sono dapprima istanziati due oggetti `File: inputFile`, che rappresenta il file originale, ed `outputFile`, che rappresenta il file copia. Subito dopo sono istanziati i relativi canali di input e di output verso questi due file, con la creazione dei due oggetti `fis` (di tipo `FileInputStream`) e `fos` (di tipo `FileOutputStream`). Infine, prima di stampare un messaggio di successo, l'applicazione esegue un ciclo `while` su tutti i byte che possono essere letti tramite l'oggetto `fis`, immagazzinandoli al volo all'interno della variabile `b`, per poi scriverli all'interno di `fos`. Si noti come il ciclo termini solo quando il metodo `read()` del `FileInputStream` restituisce il valore `-1` (ovvero quando non ci sono più byte da leggere).

In questo caso non sono stati utilizzati decoratori. Così abbiamo creato una copia binaria del file copiando i byte nella maniera più generica possibile. Questo fa sì che sia possibile specificare qualsiasi tipologia di file in input per ottenerne una copia perfetta, sia esso un file di testo, un'immagine, un documento Word o qualsiasi altro tipo di file binario.

Nel caso il file da copiare si trovi in una directory diversa da quella in cui si sta eseguendo l'applicazione, è necessario specificare il nome del file comprensivo del suo path relativo o assoluto. Un esempio di istruzione valida per eseguire la precedente applicazione mediante la specifica di un path relativo potrebbe essere il seguente:

```
java BackupFile ..\..\MioFileDacopiare
```

Un esempio che sfrutta invece un path assoluto, potrebbe essere il successivo:

```
java BackupFile C:\MiaCartella\MioFileDaCopiare
```

La versione 6 di Java ha introdotto anche altri metodi interessanti per la classe `File`:

- `setWritable(boolean writable)` consente di impostare il permesso in scrittura su un file mediante un semplice booleano.
- `setWritable(boolean writable, boolean ownerOnly)` come il metodo precedente e in più, con il secondo argomento booleano, è possibile specificare se il permesso in scrittura debba essere accordato solo all'owner del file (valore `true`) o a chiunque (valore `false`). Se il sistema operativo sottostante non è in grado di determinare l'owner del file, il permesso sarà accordato a tutti gli utenti indipendentemente dal valore dell'argomento `ownerOnly`.
- `setReadable(boolean readable)` permette di impostare il permesso di lettura di un file.
- `setReadable(boolean readable, boolean ownerOnly)` come il metodo precedente ed inoltre con il secondo argomento è possibile specificare se il permesso in lettura debba essere concesso solo all'owner del file o a chiunque. Anche in questo caso, se il sistema operativo sottostante non è in grado di determinare l'owner del file, il permesso sarà accordato a tutti gli utenti indipendentemente dal valore dell'argomento `ownerOnly`.
- `setExecutable(boolean executable)` come per i metodi `setWritable(boolean writable)` e `setReadable(boolean readable)` consente di impostare il permesso di esecuzione di un file.
- `setExecutable (boolean executable, boolean ownerOnly)` funziona come con i metodi `setWritable(boolean writable, boolean ownerOnly)` e `setReadable(boolean readable, boolean ownerOnly)`.
- `getTotalSpace()` restituisce la dimensione in byte complessiva della partizione identificata dal nome del file (per esempio `C:`). Restituisce il valore `0L` se il nome del file non coincide con una partizione valida del sistema operativo.
- `getFreeSpace()` restituisce la dimensione in byte dello spazio libero della partizione identificata dal nome del file. In particolare restituisce il numero di byte non allocati al momento della chiamata. Questo

metodo non può però prevedere se, dopo una frazione di secondo dalla sua invocazione, un altro processo (magari esterno alla Virtual Machine) andrà ad allocare nuovi byte della partizione in questione. La sua precisione è quindi spesso relativa. Anche questo metodo restituisce il valore `0L` se il nome del file non coincide con una partizione valida del sistema operativo.

- `getUsableSpace()` restituisce la dimensione in byte dello spazio realmente utilizzabile della partizione astratta dal nome del file. Questo metodo è più affidabile rispetto a `getFreeSpace()` per quanto riguarda una stima dello spazio realmente utilizzabile. Infatti controlla anche eventuali permessi in scrittura dei file ed altre restrizioni che il sistema operativo potrebbe definire. Neanche questo metodo può però prevedere se, dopo una frazione di secondo dalla sua invocazione, un altro processo andrà ad allocare nuovi byte nella stessa partizione. Anche la sua precisione è dunque spesso relativa. Ed anche questo metodo restituisce il valore `0L` se il nome del file non coincide con una partizione valida del sistema operativo.

18.4.3 Serializzazione di oggetti e modificatore `transient`

Con **serializzazione di oggetti** intendiamo il processo per rendere persistente un oggetto Java. Rendere persistente un oggetto significa far sopravvivere l'oggetto oltre la terminazione del programma. Solitamente questo significa salvare lo **stato dell'oggetto**, ovvero le variabili d'istanza con i relativi valori, all'interno di un file (o all'interno di un database, ma questo sarà argomento del prossimo capitolo). In Java è possibile serializzare oggetti a patto che implementino l'interfaccia `Serializable` del package `java.io`. Tale interfaccia non contiene metodi ma serve solo a distinguere ciò che è serializzabile da ciò che non lo è. Esistono per esempio classi della libreria standard che, per come sono concepite, non possono essere serializzate. Un esempio potrebbe essere la classe `Thread`. Un thread non ha uno stato, rappresenta un concetto dinamico, un "processo che esegue codice su dati" e non può essere serializzato. La classe `Thread`, infatti, non implementa l'interfaccia `Serializable`. Altri esempi di classi non serializzabili sono tutte

le classi di tipo `java.util.stream.Stream`.

Tenendo presente che serializzare un oggetto significa salvare il suo stato interno, ovvero salvare il valore delle proprie variabili d'istanza, se vogliamo creare una classe da cui istanziare oggetti da serializzare, bisogna stare attenti alle variabili d'istanza. Se infatti una delle variabili d'istanza è di tipo `Thread` o di tipo `Writer`, provando a serializzare l'oggetto otterremmo al runtime una `java.io.NotSerializableException`. La stessa eccezione scatterebbe nel caso in cui la nostra classe non implementasse `Serializable`. Per ovviare a questo problema esiste il modificatore `transient`. Esso può essere anteposto solo a variabili d'istanza, ed avvertirà al runtime la JVM che la variabile marcata `transient` non deve essere serializzata. È obbligatorio quindi marcare `transient` le variabili non serializzabili (per esempio di tipo `Thread`), tuttavia potremmo anche desiderare di non serializzare volutamente alcune variabili per ragioni di sicurezza. Come esercizio consideriamo la seguente semplice classe da serializzare:

```
public class Persona implements java.io.Serializable {
    private String nome;
    private String cognome;
    private transient Thread t = new Thread();
    private transient String codiceSegreto;

    public Persona(String nome, String cognome, String cs) {
        this.setNome(nome);
        this.setCognome(cognome);
        this.setCodiceSegreto(cs);
    }
    //Metodi set e get ommessi
    public String toString() {
        return "Nome: " + getNome() + "\nCognome: " +
getCognome() +
               "\nCodice Segreto: " + getCodiceSegreto();
    }
}
```

Abbiamo marcato `transient` una variabile `Thread` (eravamo obbligati) e la variabile `codiceSegreto` di tipo `String`, in quanto serializzare su un file il codice segreto potrebbe essere una mossa poco previdente. Per serializzare l'oggetto basta eseguire la seguente classe:

```
import java.io.*;

public class SerializeObject {
    public static void main(String args[]) throws
IOException {
        Persona p = new Persona("Claudio", "De Sio Cesari",
"xxx");
        try (FileOutputStream f =
               new FileOutputStream(new
File("persona.ser"));
             ObjectOutputStream s = new
ObjectOutputStream(f);) {
            s.writeObject(p);
            System.out.println("Oggetto serializzato!");
        }
    }
}
```

Il codice è semplice, non molto diverso dagli altri esempi che finora abbiamo visto. Le particolarità stanno nella decorazione eseguita con l'oggetto di tipo `ObjectOutputStream` e nell'utilizzo del metodo `writeObject()`, che come si può intuire è capace di scrivere nel flusso oggetti Java (serializzabili). Per deserializzare l'oggetto è sufficiente eseguire la seguente classe:

```
import java.io.*;

public class DeSerializeObject {
    public static void main(String args[]) throws Exception
{
        Persona p = null;
        try (FileInputStream f =
               new FileInputStream(new File("persona.ser"));
            ObjectInputStream s = new ObjectInputStream(f);)
{
          p = (Persona)s.readObject();
          System.out.println("Oggetto deserializzato!");
          System.out.println(p);
       }
    }
}
```

di cui di seguito eccone l'output:

```
Oggetto deserializzato!
```

```
Nome: Claudio
Cognome: De Sio Cesari
Codice Segreto: null
```

> **Anche le variabili dichiarate `static` non saranno serializzate. Infatti non sarebbe giusto cambiare il valore di una variabile statica, che è condivisa da tutte le istanze della classe, solo perché è possibile deserializzare una particolare istanza. Le variabili statiche sono implicitamente dichiarate `transient`.**

Attenzione! Esiste uno strano vincolo per realizzare con successo una deserializzazione. Bisogna inserire un **costruttore senza argomenti** nella prima superclasse della classe da deserializzare che non implementa `Serializable`, altrimenti andremo incontro ad una `java.io.InvalidClassException`, con un messaggio:

```
no valid constructor
```

quando sarà invocato il metodo `readObject()`.

18.4.4 Deep copy

È interessante concludere il discorso iniziato nel paragrafo 12.1.4 sulla clonazione di oggetti. Avevamo visto che il metodo `clone()` solitamente si limita a creare una *shallow copy* (copia superficiale) dell'oggetto, e che avremmo dovuto riscrivere il metodo `clone()` in maniera molto costosa se avessimo voluto ottenere una *deep copy* (copia profonda). Per gli oggetti `Serializable` però, c'è una scorciatoia interessante. Il seguente metodo:

```
 public static <O extends Serializable> O cloneObject(O
 object)
   throws IOException, ClassNotFoundException {
   if (object == null) {
       return null;
   }
   O objectCloned;
   try (ByteArrayOutputStream bout = new
```

```
ByteArrayOutputStream();
       ObjectOutputStream out = new
ObjectOutputStream(bout)) {
      out.writeObject(object);
      out.flush();
      try (ByteArrayInputStream bais =
            new ByteArrayInputStream(bout.toByteArray());
          ObjectInputStream in = new
ObjectInputStream(bais)) {
          objectCloned = (O)in.readObject();
     }
  }
  return objectCloned;
}
```

Prende in input un oggetto `Serializabile` e ne ritorna una copia profonda, in quanto essa viene creata direttamente tramite stream, e non impostando ad uno ad uno i campi. Questo tipo di implementazione inoltre non ha bisogno di essere aggiornata quando si aggiornano gli oggetti da clonare. Questo metodo statico generico, dichiara come tipo parametro `O` implementazione di `Serializable`, e lo usa come parametro del metodo. Come abbiamo detto a questo metodo si possono passare solo oggetti `Serializable`. Per quanto riguarda l'implementazione, dopo un controllo di nullità di routine, e la dichiarazione del reference `objectCloned` che dovrà puntare all'oggetto clonato restituito dal metodo, si passa al vero algoritmo. In un blocco try with resources, istanziamo un node stream `bout` di tipo `ByteArrayOutputStream`. In questo output stream è possibile immagazzinare un array di byte da sfruttare successivamente nel programma (la destinazione di questo output stream è l'oggetto stesso). Dopo istanziamo un processing stream `out` di tipo `ObjectOutputStream`, a cui passiamo come parametro (destinazione) proprio l'oggetto `bout` di tipo `ByteArrayOutputStream`. L'oggetto `out` di tipo `ObjectOutputStream`, permette di scrivere oggetti, che in questa configurazione di stream, sarà immagazzinato nel `ByteArrayOutputStream`. Infatti vengono chiamati, prima il metodo `writeObject()` che scrive nel `ByteArrayOutputStream` l'oggetto, e poi il metodo `flush()` che forza l'`ObjectOutputStream` a scaricare immediatamente il proprio buffer inviando i dati all'output stream contenuto, in questo caso `ByteArrayOutputStream`. Nel successivo blocco try with resources è eseguita

l'operazione inversa usando una concatenazione di stream di lettura formata da un oggetto `bais` di tipo `ByteArrayInputStream`, a cui viene passato come parametro (fonte) l'array di byte contenuto nell'oggetto `bout` di tipo `ByteArrayOutputStream`. Lo stesso oggetto `bais` viene poi passato in input ad un oggetto `in` di tipo `ObjectInputStream`. La chiamata al metodo `readObject()`, restituisce l'oggetto clonato.

Con Java 8 sono state introdotte due classi standard per l'encoding e il decoding (in italiano codifica e decodifica) in Base64: le classi `Base64.Encoder` e `Base64.Decoder` del package `java.util`. Stiamo parlando della codifica che trasforma un array di byte in una stringa. Questo tipo di encoding è molto usato per inserire dati binari in una email, o per passarli tramite protocolli di rete basati su stringhe come l'HTTP. Per un esempio di utilizzo consultare gli esempi da scaricare on line.

18.5 NIO 2

Nella prima parte di questo capitolo abbiamo potuto assaggiare la potenza e la complessità del package `java.io`. Tale complessità ha portato i progettisti di Java a creare nuove librerie che risolvono problemi specifici (come la gestione dei file) in modo più semplice e meno verboso. Con Java 7 sono state introdotte molte novità sulla gestione dei file. Già dalla versione 1.4 di Java fu introdotta una nuova libreria nota come **NIO** definita nel package `java.nio` **(NIO** sta per **New Input Output).** Questo definisce nuove classi e interfacce per gestire più efficacemente l'input output di file in Java. In particolare il nuovo modello proponeva l'uso di **buffer** (contenitori di dati da passare in input o ricevere in output) e **channel** (connessioni ad entità capaci di eseguire operazioni di input output). Per esempio il seguente codice legge un file di testo e lo inserisce in un `ByteBuffer` per poi rileggerlo e stamparlo a video tramite un oggetto di tipo `FileChannel`:

```
String path = "C:\\myfile.txt";
try (FileInputStream fis = new FileInputStream(path);
```

```
      FileChannel fileChannel = fis.getChannel();) {
    ByteBuffer buffer = ByteBuffer.allocate(128);
    int bytesRead = fileChannel.read(buffer);
    while (bytesRead != -1) {
       System.out.println("Letto: " + bytesRead);
       //converte il buffer da buffer di scrittura a buffer
di lettura
       buffer.flip();
       // Stampa byte letti
       while (buffer.hasRemaining()) {
           System.out.print((char) buffer.get());
       }
       buffer.clear();
       bytesRead = fileChannel.read(buffer);
   }
} catch (IOException e) {
    e.printStackTrace();
}
```

Il codice è abbastanza chiaro, visto che non si distacca molto da quello che abbiamo visto sino ad ora. I channel rispetto agli oggetti stream hanno la possibilità di gestire più situazioni, come per esempio quando si ha a che fare con file protetti o si vuole utilizzare stream non bloccanti (ovvero che non si bloccano se aspettano un input). I channel non sono stati introdotti con lo scopo di sostituire gli stream. La libreria NIO semplicemente consente operazioni più complesse che la libreria IO non permette. Il problema con la prima versione di NIO era la curva di apprendimento. Ora con **NIO 2** abbiamo anche una notevole semplicità di utilizzo. In particolare sono stati introdotti quattro nuovi package in Java 7: `java.nio.file`, `java.nio.file.spi`, `java.nio.file.attribute`, `java.nio.file.spi`. Le novità più importanti risiedono essenzialmente nell'introduzione della classe `Files` e l'interfaccia `Path` (e le sue implementazioni), che di fatto ora forniscono metodi più semplici ed efficienti per effettuare operazioni su file. Effettivamente l'utilizzo di queste nuove strutture può sostituire l'utilizzo della classe `java.io.File`. A scopo di facilitarne l'interoperabilità delle librerie (e volendo la migrazione) la classe `File` definisce il metodo `toPath()` per ottenere un oggetto `Path`. Esiste anche l'omologo metodo `toFile()` nell'interfaccia `Path`.

18.5.1 L'interfaccia `Path`

`Path` rappresenta un percorso sul file system che astrae il concetto di file o di directory. Un **path** (in inglese **percorso)** è un concetto familiare a chi usa il computer. Un tipico esempio di `Path` potrebbe essere: C:\Windows\write.exe che rappresenta il programma WordPad su Windows. Su Linux per esempio un `Path` è qualcosa di diverso visto il differente file system: /etc/grub.conf identifica il file di configurazione del programma Grub. Questa è anche la ragione per cui `Path` è un'interfaccia e non una classe. Per ottenere un'implementazione di `Path` infatti bisogna utilizzare l'oggetto `FileSystem`:

```
Path path1 =
FileSystems.getDefault().getPath("/root/aFile.txt");
```

o più semplicemente l'oggetto `Paths`:

```
Path path2 = Paths.get("C:\\Program Files\\EJE");
```

> **Il doppio backslash \\ è obbligatorio visto che uno solo serve per identificare i caratteri di escape.**

Una volta ottenuto un oggetto `Path` è possibile utilizzare una serie di metodi potenti e semplici da utilizzare. Segue qualche esempio:

```
Path pathToDesktop = Paths.get("C:\\Users\\user\\Desktop");
Path pathToDocuments =
Paths.get("C:\\Users\\user\\Documents");
System.out.println("toString: " + pathToDesktop.toString());
System.out.println("getFileName: " +
pathToDesktop.getFileName());
System.out.println("getName(0): " +
pathToDesktop.getName(0));
System.out.println("getNameCount: " +
pathToDesktop.getNameCount());
System.out.println("subpath(0,2): " +
pathToDesktop.subpath(0,2));
System.out.println("getRoot: " + pathToDesktop.getRoot());
System.out.println("getParent: " +
pathToDesktop.getParent());
System.out.println("toUri: " + pathToDesktop.toUri());
```

```
 System.out.println("path from p1 to p2: " +
    pathToDesktop.relativize(pathToDocuments));
 System.out.println("path from p2 to p1 :  " +
    pathToDocuments.relativize(pathToDesktop));
 System.out.println("pathToDesktop.equals(pathToDocuments): "
 +
    pathToDesktop.equals(otherPath));
 System.out.println("pathToDesktop.startsWith: " +
    pathToDesktop.startsWith(pathToDocuments.subPath(0,2)));
 System.out.println("pathToDesktop.endsWith: " +
    pathToDesktop.endsWith(pathToDocuments.subPath(0,2))));
```

Segue l'output del codice presentato:

```
toString: C:\Users\user\Desktop
getFileName: Desktop
getName(0): Users
getNameCount: 3
subpath(0,2): Users\user
getRoot: C:\
getParent: C:\Users\user
toUri: file:///C:/Users/user/Desktop
path from p1 to p2: ..\Documents
path from p2 to p1: ..\Desktop
pathToDesktop.equals(pathToDocuments): false
pathToDesktop.startsWith: false
pathToDesktop.endsWith: false
```

Alcuni di questi metodi non hanno bisogno di ulteriori commenti, quindi commenteremo solo i più enigmatici. Il metodo `subpath()` per esempio restituisce il numero di elementi che compongono il path, escludendo il nodo radice (nell'esempio è `C:\` ma per un sistema Unix-Linux sarebbe `/`). Il metodo `getRoot()` serve proprio per identificare il nodo radice. Il metodo `getParent()` invece restituisce la directory in cui è contenuto il file rappresentato dall'oggetto `Path`. Il metodo `toUri()` restituisce il path in formato **URI (Uniform Resource Identifier).** Interessante il metodo `relativize()` che restituisce il percorso che serve per andare dal path su cui si chiama il metodo al path che viene passato come parametro. Nell'esempio il metodo `startsWith()` ritorna `false` in quanto il metodo `subpath()` non restituisce il nodo radice.

18.5.2 La classe `Files`

Per compiere operazioni sui file la classe principale di riferimento ora si chiama `Files` (come `Path` è localizzata nel package `java.nio.file`). Esistono tantissimi metodi di utilità nella classe `Files` (sono oltre cinquanta escludendo quelli ereditati da `Object`, tutti statici). Anche in questo caso i metodi sono piuttosto intuitivi. Per esempio per ottenere un `BufferedWriter` per scrivere su un file avremo a disposizione il metodo `newBufferedWriter()`. Ecco un esempio che fa uso del costrutto try with resources:

```
Charset charset = Charset.forName("UTF-8");
String contenutoDelFile = "Ciao";
Path path = Paths.get("C:\\Users\\user\\Desktop\\test.txt");
try (BufferedWriter writer = Files.newBufferedWriter(path,
charset)) {
     writer.write(contenutoDelFile, 0,
contenutoDelFile.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Come per l'interfaccia `Path`, diamo uno sguardo ai metodi più interessanti della classe `Files`:

```
Path directory = Paths.get("C:\\Users\\user\\Desktop");
Path file = Paths.get("C:\\Users\\user\\Desktop\\test.txt");
System.out.println("Files.exists(directory): " +
Files.exists(directory));
System.out.println("Files.isReadable(file): " +
Files.isReadable(file));
System.out.println("Files.isWritable(file): " +
Files.isWritable(file));
System.out.println("Files.isExecutable(file): " +
Files.isExecutable(file));
System.out.println("Files.isSameFile(file): " +
  Files.isSameFile(directory, file));
```

e di seguito l'output:

```
Files.exists(directory): true
Files.isReadable(file): true
Files.isWritable(file): true
```

```
Files.isExecutable(file): true
Files.isSameFile(file): false
```

Come si può immaginare il metodo `exists()` controlla se il file specificato esiste fisicamente; solleva una `SecurityException` nel caso il file non sia accessibile. Esiste anche il metodo `notExists()`. I metodi `isReadable()`, `isWritable()` e `isExecutable()` servono rispettivamente per capire i permessi di leggibilità, scrittura ed esecuzione del file. Il metodo `isSameFile()` invece restituisce `true` se e solo se i file specificati come parametri puntano allo stesso file (cosa possibile per esempio in presenza di file di tipo collegamento). I metodi che consentono operazioni sui file sono estremamente intuitivi nonostante siano anche molto potenti. Per l'eliminazione di un file esistono due metodi: `delete()` e `deleteIfExists()`. Entrambi definiscono come parametro in input un oggetto `Path` che rappresenta il file da eliminare. La differenza tra questi due metodi è che il secondo non solleva eccezioni nel caso l'eliminazione fallisca. Il metodo `delete()` invece potrebbe fallire nel caso ci sia un problema con i permessi di eliminazione del file (`IOException`), se si provasse a eliminare un file che non esiste (`NoSuchFileException`) oppure se si provasse a eliminare una directory non vuota (`DirectoryNotEmptyExcept ion`). Per quanto riguarda la copia di file il metodo `copy()` prende in input due parametri `Path` (il primo è il `Path` che si deve copiare mentre il secondo è quello di destinazione) e un varargs di tipo `CopyOption`. Quest'ultima è un'interfaccia implementata da due enumerazioni `StandardCopyOption` e `LinkOption`. Ecco un semplice esempio:

```
import static java.nio.file.StandardCopyOption.*;
import java.nio.file.LinkOption;
//...
Path source =
Paths.get("C:\\Users\\user\\Desktop\\test.txt");
Path target =
Paths.get("C:\\Users\\user\\Desktop\\copy.txt");
Files.copy(source, target, REPLACE_EXISTING,
COPY_ATTRIBUTES,
  LinkOption.NOFOLLOW_LINKS);
```

che crea una copia del file test.txt chiamandola copy.txt. Delle tre opzioni

specificate come varargs, i primi due sono elementi importati staticamente dall'enumerazione `StandardCopyOption`, mentre la terza è un elemento dell'enumerazione `LinkOption`. `REPLACE_EXISTING` implica che se il file copy.txt esistesse già, sarà sovrascritto. `COPY_ATTRIBUTES` copia anche gli attributi del file test.txt; per esempio se il file test.txt è in sola lettura anche il file test.txt sarà in sola lettura. Infine `NOFOLLOW_LINKS` farà in modo che, nel caso il file test.txt fosse un collegamento ad un file, non sia copiato il file vero e proprio ma solo il file collegamento. Infatti di default questo metodo avrebbe copiato il file di destinazione e non il link. Queste `CopyOption` sono le uniche supportate dal metodo `copy()`.

È possibile copiare directory ma i file contenuti in essa non saranno copiati. È bene notare che esistono anche due overload del metodo `copy()` che interagiscono con `InputStream` e `OutputStream`. Volendo quindi è possibile usare un `OutputStream` per scrivere una directory dopo averla letta.

Il discorso è molto simile per lo spostamento dei file. Anche il metodo `move()` ha la stessa firma di parametri di `copy()`. La differenza sta nel fatto che `move()` supporta solo gli elementi delle enumerazioni `StandardCopyOption.REPLACE_EXISTING` che abbiamo già descritto, e `StandardCopyOption.ATOMIC_MOVE`. Quest'ultimo serve a garantire che lo spostamento sia eseguito come operazione atomica a tutte quelle che si possono collegare.

La classe `Files` contiene anche una quindicina di metodi per gestire gli attributi e i metadati dei file. Per esempio `isHidden()` restituisce `true` se il `Path` specificato in input è un file nascosto, mentre `size()` restituisce la dimensione del `Path` specificato. Ci sono metodi per leggere gli attributi (`getAttribute()` e `readAttributes()`), per scrivere gli attributi (`setAttribute()`), per conoscere o impostare la data dell'ultima modifica (`getLastModifiedTime()` e `setLastModifiedTime()`), conoscere o impostare l'autore (`getOwner()` e `setOwner()`), etc. Infine la classe `Files` definisce

diversi metodi per scrivere e leggere file. Molti di questi prendono in input come parametro opzionale uno o più elementi dell'enumerazione `StandardOpenOption`. Ecco l'elenco e la descrizione dei vari elementi:

- `WRITE`: apre il file con accesso in scrittura.
- `APPEND`: aggiunge nuovi dati alla fine del file. Questa opzione viene usata in congiunzione alle opzioni `WRITE` o `CREATE`.
- `TRUNCATE_EXISTING`: tronca il file a zero byte. Questa opzione viene usata in congiunzione all'opzione `WRITE`.
- `CREATE_NEW`: crea un nuovo file e solleva un'eccezione se il file esiste già.
- `CREATE`: apre il file se esiste già o lo crea se non esiste.
- `DELETE_ON_CLOSE`: cancella il file quando lo stream viene chiuso. Utile nel caso di file temporanei.
- `SPARSE`: suggerisce (visto che è un'operazione che dipende dal file system) che il file appena creato sia in formato sparso. Questa opzione avanzata è supportata su alcuni filesystem come NTFS, dove file di grandi dimensioni sono memorizzati in aree di memoria non contigue che sono comunque disponibili per l'utilizzo.
- `SYNC`: mantiene il file (sia di contenuti e metadati) sincronizzato con il dispositivo di storage sottostante.
- `DSYNC`: mantiene solo il contenuto del file sincronizzato con il dispositivo di storage sottostante.

Metodi adatti a leggere file di piccole dimensioni sono `readAllBytes()` per leggere file binari, e `readAllLines()` per leggere file di testo. L'equivalente metodo per la scrittura è `write()`, di cui esistono vari overload per prendere in input o un array di byte, o un oggetto che implementi una collezione o un array di caratteri (o stringhe). Sia quando si legge che quando si scrive un file di testo, bisogna specificare anche un oggetto `Charset` per la decodifica (per esempio `Charset.forName("UTF-8")`).

Mediante un oggetto `Files` possiamo ottenere anche

`BufferedInputReader` e `BufferedWriter` utilizzando i metodi `newBufferedInputReader()` e `newBufferedWriter()`, come abbiamo già visto:

```
String s = "Java 7";
try (BufferedWriter writer = Files.newBufferedWriter(file,
       Charset.forName("UTF-8"))) {
    writer.write(s, 0, s.length());
}
```

Inoltre è possibile anche creare stream più semplici per poi decorarli come abbiamo visto precedentemente:

```
import static java.nio.file.StandardOpenOption.*;

Path file = Paths.get("C:\\Program
Files\\EJE\\LEGGIMI.htm");
String firma = "Creato da Claudio...";
byte data[] = firma.getBytes();

try (OutputStream out =
  new
BufferedOutputStream(Files.newOutputStream(file,CREATE,
APPEND))) {
    out.write(data, 0, data.length);
}
```

Il codice precedente aggiunge una firma al file specificato.

È possibile anche creare oggetti della libreria NIO come `ByteChannel` con il metodo `newByteChannel()`, per poi poterli utilizzare sfruttando oggetti `ByteBuffer`. È semplice creare file temporanei con le seguenti istruzioni:

```
try {
    Path fileTemporaneo = Files.createTempFile(null,
".tmp");
    System.out.format("Creato il file temporaneo: " +
fileTemporaneo);
}
```

Con questo codice abbiamo creato un file temporaneo senza specificarne il nome, ma specificandone il suffisso (“.tmp”), e poi abbiamo stampato il suo nome.

Il percorso di salvataggio dipende dal sistema operativo.

Segue l'output:

```
Creato il file temporaneo:
  C:\Users\user\AppData\Local\Temp\2041268323571170425.tmp
```

Il metodo `lines()` della classe `Files`, che abbiamo già visto con la classe `BufferedReader`, ritorna uno `Stream<String>`. Inoltre `Files` definisce un metodo che si chiama `list()` e che ritorna uno `Stream<Path>` di file presenti nella directory specificata. Per esempio con il seguente codice:

```
Path pathToDirectory = Paths.get(".");
try (Stream<Path> files = Files.list(pathToDirectory)) {
    files.filter(p ->
p.getFileName().toString().endsWith(".java")).
       forEach(System.out::println);
}
```

stamperemo tutti i file con suffisso .java della directory corrente. Il metodo `list()` però si limita ai file della directory indicata. Volendo scendere all'interno delle directory contenute nella directory specificata, è possibile usare il metodo `walk()` che prende in input il path e la profondità ("depth" in inglese) che indica il livello di profondità in cui si deve scendere nell'albero delle directory. Se poi vogliamo anche accedere agli attributi dei file è necessario utilizzare il metodo `find()`. Per esempio con il seguente frammento di codice stampiamo tutti i file creati nella directory corrente nell'ultima mezz'ora.

```
Path directory = Paths.get(".");
Instant ultimaMezzora = Instant.now().minus(30,
ChronoUnit.MINUTES);
try (Stream<Path> files = Files.find(directory, 1,
  (path, attributi) -> attributi.creationTime().toInstant()
  .compareTo(ultimaMezzora) >= 0)) {
    files.forEach(System.out::println);
}
```

Riepilogo

Il package **`java.io`** è molto vasto ed è organizzato principalmente seguendo il **pattern Decorator.** Tale pattern permette la cosiddetta ereditarietà dinamica, ovvero organizzando le classi secondo la struttura del pattern, si riescono a creare oggetti da decorare con nuove caratteristiche, senza che queste caratteristiche debbano essere ereditate da una superclasse. Una volta compreso come funziona il pattern, il package `java.io` dovrebbe essere più semplice da utilizzare.

Il concetto di **stream (flusso)** dell'input-output è diverso da quello che abbiamo studiato nel capitolo precedente. Essenzialmente gli stream rappresentano flussi di canali di dati personalizzati. Gli stream sono categorizzati in diversi modi. In primo luogo esistono stream di input e stream di output. Gli **stream di input** sono oggetti che si connettono ad una **fonte** da cui leggere (una stringa, un indirizzo di rete, un array di byte, un file, etc.) e permettono la lettura dei dati dalla fonte mediante metodi **`read()`**. Gli **stream di output** sono oggetti che si connettono ad una destinazione dove scrivere tramite metodi **`write()`**. Un'altra categorizzazione degli stream consiste nel dividere **byte stream** e **character stream**, ovvero stream (di input o di output) che agiscono su unità di tipo `byte` (8 bit), o di tipo `char` (16 bit). Le super interfacce principali sono i byte stream **`InputStream`** e **`OutputStream`**, e i character stream **`Reader`** e **`Writer`.** InputStream e `Reader` definiscono praticamente gli stessi metodi, con la differenza che `InputStream` usa i byte come unità di lettura, mentre `Reader` usa i caratteri. Stesso discorso vale per le interfacce `OutputStream` e `Writer`, che per esempio dichiarano rispettivamente **`write(byte b)`** e **`write(char c).`** Infine abbiamo distinto i **node stream** dai **processing stream.** I **node stream** di input possono leggere da una fonte direttamente tramite i metodi `read()`, così come un node output stream può scrivere su una destinazione tramite i metodi `write()`. Esempi sono le classi **`FileInputStream`** e **`FileOutputStream`.** I **processing stream** sono dei decoratori, ovvero classi che prendono in input altri stream (o processing o node stream), e solitamente hanno il ruolo di fornire metodi per leggere o scrivere in maniera più semplice o efficiente. Esempi di processing stream sono le classi **`ObjectInputStream`** e **`ObjectOutputStream`.**

Gli stream vengono aperti automaticamente quando sono istanziati, poi dopo aver letto da una fonte o scritto in una destinazione devono essere chiusi con il metodo **`close()`**. L'alternativa consigliata è l'utilizzo del costrutto **try with resources.** Uno dei punti deboli della libreria oltre la complessità di utilizzo è rappresentato dalla verbosità del codice. Le **checked exception** sono dichiarate nella maggior parte dei metodi e costruttori dichiarati nella libreria.

Per leggere quanto viene scritto da tastiera è possibile utilizzare più soluzioni. Per esempio concatenare un **`BufferedReader`** ad un **`InputStreamReader`** la cui fonte è l'oggetto **`System.in.`** BufferedReader ci offre diversi metodi di tipo `read()`, come `readLine()`, che legge una riga alla volta. Anche la classe **`Scanner`** può adempiere allo stesso scopo.

La classe **`File`** dichiara decine di metodi per gestire file e directory, ma rappresenta pur sempre una fonte o una destinazione, e non uno stream. Quindi per leggere e scrivere file sono comunque necessarie le classi **`FileInputStream`** e **`FileOutputStream`**, oppure **`FileWriter`** e **`FileReader`** nel caso di file testuali. La **serializzazione di oggetti** è il processo per rendere persistente un oggetto Java, ovvero far sopravvivere un oggetto oltre la chiusura del programma. Ciò implica salvare lo **stato dell'oggetto**, ossia le variabili d'istanza con i relativi valori, all'interno di un file. Un oggetto per essere serializzabile deve implementare l'interfaccia (che non contiene metodi) **`Serializable;`** inoltre anche le sue variabili d'istanza devono essere serializzabili. Nel caso non lo fossero (nel caso di classi come `Thread`) o intenzionalmente non si voglia salvare lo stato di qualche variabile, bisogna dichiararle **`transient.`** Tale modificatore può essere usato solo davanti a variabili d'istanza. Tante classi della libreria standard implementano già `Serializable` (come per esempio la classe `String`).

Possiamo eseguire una **deep copy** di un oggetto concatenando un **`ByteArrayOutputStream`** ad un **`ObjectOutputStream`**, ed utilizzando il metodo **`writeObject()`** di quest'ultimo per scrivere l'oggetto nel `ByteArrayOutputStream`. Poi concatenando un **`ByteArraylnputStream`** ad un **`ObjectInputStream`**, si può utilizzare il metodo **`readObject()`** di quest'ultimo per leggere l'oggetto dal `ByteArraylnputStream`.

Già dalla versione 1.4 di Java fu introdotta una nuova libreria nota come **NIO** definita nel package `java.nio`. Questa propone l'uso di **buffer**

(contenitori di dati da passare in input o ricevere in output) e **channel** (connessioni ad entità capaci di eseguire operazioni di input output). La libreria NIO semplicemente consente operazioni più complesse che la libreria IO non permette, ma rimane una libreria complessa. La libreria **NIO 2** invece, è caratterizzata da una notevole semplicità di utilizzo. Essenzialmente bisogna conoscere la classe **`Files`** e l'interfaccia **`Path`**, che forniscono metodi più semplici ed efficienti per effettuare operazioni su file (che in `java.io` sono delegate alla classe `File` e agli stream sui file). Le librerie possono cooperare, infatti la classe `File` definisce il metodo **`toPath()`** per ottenere un oggetto `Path`, e l'interfaccia `Path` definisce il metodo **`toFile()`**. **`Path`** rappresenta un percorso sul file system che astrae il concetto di file o di directory. Il modo più semplice per ottenere un oggetto `Path`, è quello di utilizzare il metodo statico **`get()`** della classe **`Paths.`** La classe **`Files`** infine, dichiara decine di metodi statici per compiere operazioni sui file.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Aver compreso il pattern Decorator (unità 18.1, 18.2)	☐	
Saper riconoscere nelle classi del package `java.io` i ruoli definiti nel pattern Decorator (unità 18.3)	☐	
Capire le fondamentali gerarchie del package `java.io` (unità 18.3)	☐	
Avere confidenza con i tipici problemi che si incontrano con l'input-output, come la serializzazione degli oggetti e la gestione dei file (unità 18.4)	☐	
Avere confidenza con il nuovo modello per l'input output di Java 7 denominato NIO 2 (unità 18.5)	☐	

Note:

19

Moduli

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✔ Comprendere le ragioni che hanno portato all'introduzione del nuovo sistema modulare di Java 9 (unità 19.1).
- ✔ Saper padroneggiare i moduli e le relative direttive (unità 19.2, 19.3).
- ✔ Saper implementare servizi basati sui moduli per eliminare l'accoppiamento tra sottosistemi (unità 19.3).
- ✔ Saper creare JAR modulari e file JMOD utilizzando gli strumenti del JDK (unità 19.4).
- ✔ Comprendere e saper affrontare le difficoltà per migrare applicazioni scritte in passato a Java 9 (unità 19.5).
- ✔ Saper creare runtime personalizzati con lo strumento jlink (unità 19.6).

In questo capitolo parleremo della più importante novità di Java 9: il **Java Platform Module System** (in italiano **sistema modulare della piattaforma Java).** Esso introduce il **modulo**, un nuovo componente di programmazione che si può definire come un contenitore auto-descrittivo di codice e dati. Fisicamente un modulo è

semplicemente una cartella che contiene una classe che descrive il modulo, oltre a package ed eventualmente altri tipi di risorse.

Il sistema modulare introduce inoltre un'altra serie di importanti novità come lo strumento **jlink**. Esso può essere utilizzato tra la fase di compilazione e la fase di esecuzione del programma, per assemblare un insieme di moduli ottimizzato per il runtime. In questo modo possiamo eseguire i nostri programmi senza essere obbligati come in passato a portarci dietro librerie che non utilizziamo, migliorando la qualità del nostro software in termini di performance e dimensioni.

Con l'introduzione dei moduli, oltre ai classici Java Archive (file JAR), ora è possibile creare anche file **JAR modulari**, ovvero file JAR contenenti moduli. È stato introdotto anche il nuovo formato **JMOD**, un file simile ad un file JAR ma che può includere codice nativo e file di configurazione. Esso si può generare tramite il nuovo strumento **jmod**. Le stesse librerie del JDK sono state riviste e organizzate all'interno di moduli impacchettati in archivi jmod e posti nella cartella jmods del JDK. Ma la cosa più rilevante è che con Java 9 è possibile creare il cosiddetto **forte incapsulamento**, ovvero rendere completamente inaccessibili alcuni package e tipi contenuti all'interno dei moduli.

> **In questo capitolo utilizzeremo il termine "tipo" per indicare indistintamente classi, interfacce, enumerazioni ed annotazioni Java.**

Il concetto di **servizio** tramite la API **Service Loader** già esistente a partire da Java 6, è stato rivisto per essere utilizzato con la modularizzazione, e consente di implementare il principio di inversione della dipendenza con grande efficacia. Il progetto **Jigsaw** (che in italiano possiamo tradurre come "puzzle") che teorizzava una piattaforma modulare, fu proposto già nel 2005, e all'epoca era previsto che fosse realizzato già in Java 7. Dopo tantissimi rinvii e la ridefinizione di Jigsaw sotto forma di altri sotto progetti (per dettagli visitare la pagina del progetto http://openjdk.java.net/projects/jigsaw), finalmente la modularità è diventata realtà in Java 9. L'argomento potrebbe risultare ostico e non molto

interessante per alcuni lettori. Probabilmente in una realtà aziendale la definizione della struttura dei moduli di un'applicazione dovrebbe essere demandata all'architetto, il quale si occupa dei requisiti non funzionali. Ciononostante conoscere come funzionano i moduli è ormai una necessità, e comprendere bene questo argomento può risultare determinante per la propria carriera lavorativa. Per tale ragione in questo capitolo sono introdotti anche alcuni concetti architetturali e di progettazione per supportare lo studio del sistema modulare.

19.1 Perché un sistema modulare?

Era necessario introdurre un nuovo componente di programmazione, cambiare la struttura e riorganizzare in moduli l'intero JDK, creare nuovi formati e nuovi strumenti di utilità e modificare quelli esistenti? Quali problemi risolverà il Java Platform Module System? Grazie ad esso i problemi che possono essere risolti sono molteplici, e riguardano sia lo sviluppo che la distribuzione delle applicazioni Java.

19.1.1 Pro

Il nuovo modello modulare porterà i seguenti vantaggi.

- **Configurabilità affidabile** (in inglese **reliable configuration**): la modularità fornisce meccanismi per dichiarare esplicitamente le dipendenze tra i moduli non solo in fase di esecuzione, ma anche in fase di compilazione. Questo è molto importante perché con il vecchio meccanismo dei file JAR e del classpath (cfr. appendice E), in fase di esecuzione non è così raro incappare in eccezioni bloccanti come la `ClassNotFoundException`. Ora invece possiamo risolvere il problema delle dipendenze per determinare il sottoinsieme di tutti moduli necessari per supportare la nostra applicazione, prima che essa sia eseguita.
- **Forte incapsulamento** (in inglese **strong encapsulation**): i package contenuti in un modulo sono inaccessibili ad altri moduli. Bisogna esportali esplicitamente per renderli accessibili. È possibile rendere accessibile un package anche solo a determinati moduli. Inoltre, i

moduli che vogliono accedere ad un certo package contenuto in un altro modulo, devono esplicitamente richiedere tali package al modulo che li contiene. Questo forte incapsulamento dei package permette di creare progetti più ordinati, dove il concetto di package viene valorizzato, e l'accessibilità ai tipi dichiarati rimane completamente sotto controllo. Questo porta anche a migliorare la sicurezza delle nostre applicazioni, perché è possibile negare l'accesso a classi cruciali a potenziali aggressori.

- **Piattaforma Java scalabile** (in inglese **scalable Java platform**): sino a Java 8 la piattaforma Java era costituita da un unico blocco monolitico comprendente un enorme numero di package altamente dipendenti tra loro. Negli anni il JDK è cresciuto a dismisura sino a quando si è resa necessaria una riorganizzazione drastica. Inoltre la libreria Java era contenuta in un unico file chiamato rt.jar, che era necessario per eseguire qualsiasi programma. Quindi, supponendo che la nostra applicazione non utilizzi database ed interfacce grafiche, fino a Java 8 avremmo dovuto comunque utilizzare l'intero rt.jar che conteneva tutte le librerie, anche quelle grafiche e relative ai database, e la sua dimensione è aumentata sempre più. Nella versione 9.0.1 la piattaforma Java è invece costituita da 98 moduli, ma in futuro questo numero crescerà. Ora è possibile creare runtime personalizzati composti dai soli moduli necessari all'esecuzione della propria applicazione. Quindi si può ora facilmente ottimizzare la dimensione del runtime Java.

- **Prestazioni migliori:** la modularizzazione e la personalizzazione del runtime Java, porteranno come conseguenza il miglioramento delle prestazioni. Infatti le tecniche di ottimizzazione della JVM sono più efficaci nel caso siano noti a priori i tipi che saranno utilizzati nell'applicazione.

- **Maggiore integrità della piattaforma:** sino a Java 8 i programmatori hanno spesso utilizzato classi che non erano state concepite per essere utilizzate da noi sviluppatori. Ci riferiamo alle classi dei package `sun.*`, e `*.internal.*` come per esempio `sun.misc.Unsafe`, che sono utilizzate dal JDK ma essendo dichiarate pubbliche erano

accessibili anche alle nostre applicazioni. Con il forte incapsulamento introdotto dai moduli, tali librerie sono diventate *quasi* inaccessibili. Infatti fino a quando non esisteranno librerie sostitutive equivalenti e sicure, per la compatibilità all'indietro che da sempre caratterizza le nuove versioni di Java, queste librerie saranno ancora utilizzabili in qualche modo. Questo perché se non lo fossero, alcuni dei software più utilizzati in assoluto come Spring non sarebbero più utilizzabili.

19.1.2 Contro

Uno svantaggio della maggiore integrità della piattaforma è quindi proprio la **migrazione** dei programmi esistenti a Java 9, di cui ci occuperemo esplicitamente nel paragrafo 19.3.

Inoltre, se da un lato la modularizzazione risolverà diversi problemi, il **concepimento delle applicazioni Java** si è complicato. Il concetto di modulo è diventato fondamentale, e non creeremo più applicazioni vere senza dichiarare moduli. Il ciclo di sviluppo quindi richiede maggior attenzione all'architettura, dei passaggi in più per gestire il forte incapsulamento, la generazione di un runtime personalizzato e così via. Il concetto di package ora è più importante, perché i moduli espongono package e non tipi, e quindi bisogna progettarli bene (attività che, per esperienza diretta, possiamo dire essere considerata sempre secondaria).

19.2 Definizione di modulo

Come sappiamo i package rappresentano un modo per raggruppare tipi (ovvero classi, interfacce, enumerazioni ed annotazioni Java) all'interno di cartelle che formano gerarchie. Un **modulo** è una cartella che contiene un gruppo di package relazionati tra loro, opzionalmente altre risorse come immagini e file di configurazione (che per convenzione sono posti in una cartella a parte chiamata res), e un **descrittore del modulo** (in inglese **module descriptor).** Quest'ultimo è definito come un file Java compilabile denominato **module-info.java**, che una volta compilato nel file **module-info.class**, deve essere posizionato alla radice delle cartelle del modulo. La sintassi di questo file è completamente diversa da quelle degli altri tipi compilabili di Java (classi, interfacce, annotazione ed enumerazioni). Come

abbiamo già accennato nel capitolo 2, la prima anomalia riguarda le parole `module`, `open`, `opens`, `provides`, `requires`, `to`, `transitive`, `uses` e `with`, che all'interno della dichiarazione del descrittore del modulo diventano keyword (a seconda della posizione in cui vengono usate), e non possono essere utilizzate come identificatori. Sono infatti dette **parole a utilizzo limitato** (in inglese **restricted word).** La sintassi del descrittore del modulo è così definita:

```
[open] module nome_del_modulo {
    [module_directive_1;]
    [module_directive_2;]
    ...
    [ module_directive_n;]
}
```

Dove `open` è l'unico modificatore (opzionale) che possiamo utilizzare nella dichiarazione di un modulo e che spiegheremo nel paragrafo 19.2.2.3. Il `nome_del_modulo` segue la stessa convenzione dei nomi dei package (sempre lettere minuscole separate con punti).

Un modulo, come un package, coincide con una cartella. A differenza dei package però un modulo è costituito da un'unica cartella, con un nome che contiene più parole separate da un punto. Per esempio il modulo `java.base` più volte nominato nei precedenti capitoli, è una cartella che si chiama proprio `java.base`, e non una cartella chiamata `base` contenuta in un'altra cartella chiamata `java`. Solitamente la prima parola dell'identificatore del modulo (quella che precede il punto) coincide con il primo package contenuto in esso (per esempio `java.base` contiene package come `java.lang`, `java.util` e così via). Un'altra convenzione largamente utilizzata per i nomi dei package (e quindi dei moduli) è quella di utilizzare l'"inversione del nome del dominio" (in inglese "reverse domain name"). Essa consiste nel capovolgere il nome del nostro dominio aziendale ed usarlo come nome. Per esempio le applicazioni del dominio claudiodesio.com **dovrebbero usare come nome del modulo**

base com.claudiodesio. **Stesso discorso per i package . . . radice che dovrebbero sempre cominciare con** com.claudiodesio. **I nomi dei package possono anche coincidere con quelli dei moduli. Tuttavia utilizzeremo questa convenzione solo nella seconda parte del capitolo.**

Infine le `module_directive` (anch'esse opzionali) rappresentano **istruzioni per il modulo.** Possono essere di cinque tipi, ognuno dei quali contiene una delle seguenti parole chiave: `opens`, `provides`, `requires`, `exports`, `uses`. Tali direttive servono essenzialmente per gestire l'utilizzo dei tipi contenuti nei package dichiarati in un modulo, da parte di un altro modulo. Prima di entrare nei dettagli di tali direttive però, tocchiamo con mano il concetto di modulo realizzando la nostra prima applicazione modulare.

È possibile utilizzare alcune annotazioni con i moduli, come `Deprecated` **e** `SuppressWarning`.

19.2.1 HelloModularWorld

Nei seguenti esempi scriveremo ed eseguiremo i programmi da riga di comando. Come per la "gestione a mano" dei package (cfr. capitolo 6.6.1) la complessità dei comandi da eseguire potrebbe rendere l'argomento più complicato. Tuttavia è possibile utilizzare un IDE pronto per Java 9 come IntelliJ Idea, Eclipse (con un plugin per il supporto di Java 9 scaricabile a questo indirizzo https://marketplace.eclipse.org/content/java-9-support-beta-oxygen), **oppure potete scaricare una versione provvisoria della prossima versione di Netbeans a questo indirizzo** http://wiki.netbeans.org/JDK9Support. **Gli IDE gestiscono i comandi e il posizionamento dei file automaticamente, e l'utilizzo dei moduli si semplificherà molto.**

In questo paragrafo creeremo il nostro primo semplicissimo modulo. Visto che ci muoveremo da riga di comando senza un IDE, cerchiamo di seguire dei passi precisi.

Posizioniamoci in una cartella qualsiasi con la prompt di DOS (per esempio nella cartella C:/java9/capitolo_19/hello-modular-world). Chiameremo questa cartella, **cartella radice.** Al suo interno creiamo due cartelle: src che conterrà i nostri file sorgente, e la cartella mods che conterrà il contenuto della compilazione. Nella cartella src quindi creiamo una sottocartella con lo stesso nome che daremo al nostro modulo: cdsc.mymodule. Questa cartella rappresenta in effetti fisicamente il modulo stesso. Quindi qui vanno piazzati i sorgenti del modulo e il file descrittore. I nostri sorgenti sono costituiti da un'unica classe che stamperà la stringa `Hello Modular World!`:

```
package java9.cap19;

public class HelloModularWorld {
    public static void main(String[] args) {
        System.out.println("Hello Modular World!");
    }
}
```

HelloModularWorld.java deve essere posizionata all'interno della gerarchia di cartelle definite come package, ovvero nella cartella C:/java9/capitolo19/hello-modular-world/src/cdsc.mymodule/java9/cap19.
All'interno di cdsc.mymodule posizioniamo anche il seguente module descriptor (module-info.java):

```
module cdsc.mymodule {
}
```

Essendo posizionati con la prompt di DOS nella cartella radice C:/java9/capitolo19/hello-modular-world, possiamo utilizzare il seguente comando per compilare il suo descrittore e la classe HelloModularWorld.java, creando il modulo cdsc.mymodule nella cartella mods:

```
javac -d mods/cdsc.mymodule src/cdsc.mymodule/java9/cap19/*
  src/cdsc.mymodule/module-info.java
```

Questo comando creerà nella sottocartella mods la cartella cdsc.mymodule che rappresenta il modulo. All'interno di essa posizionerà il descrittore (compilato) module-info.class, e i package contenenti HelloModularWorld.class, come mostrato in figura 19.1. Infatti l'opzione -d (che sta per -directory) e che abbiamo già incontrato nei paragrafi 2.6.2 e 6.6.1, fa sì che siano create automaticamente le directory necessarie. Il primo parametro mods/cdsc.mymodule invece sta ad indicare come si chiama il modulo che vogliamo creare. In questo caso esso sarà chiamato cdsc.mymodule e sarà posizionato in una sottocartella mods che verrà creata al volo.

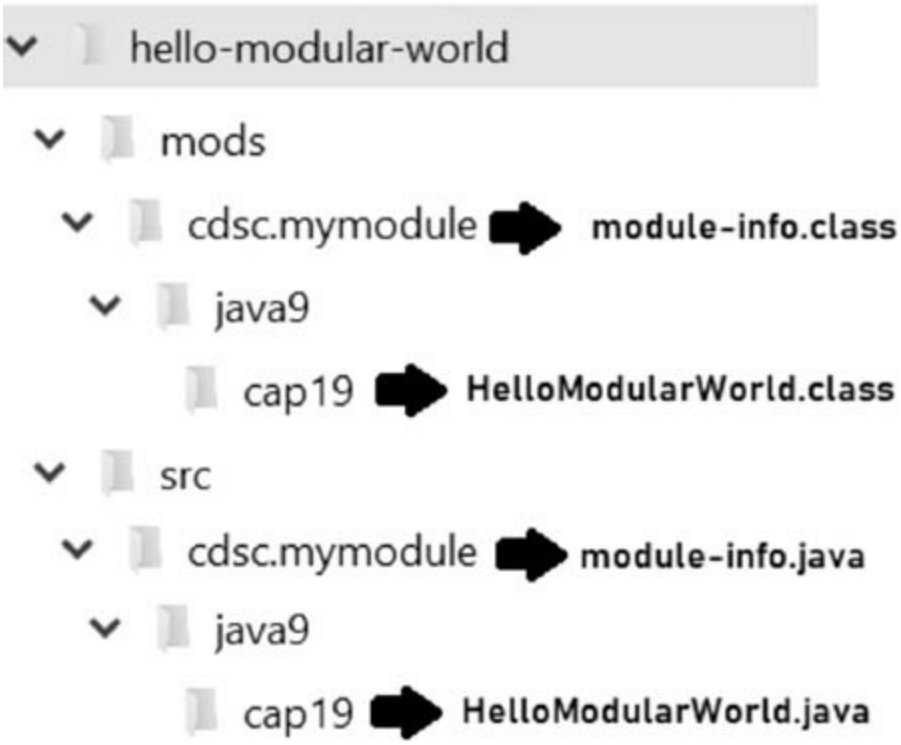


Figura 19.1 - Posizionamento dei file .java e .class.

Con il secondo parametro src/cdsc.mymodule/java9/cap19/* stiamo chiedendo al compilatore di compilare tutte le classi all'interno della cartella specificata (nel nostro caso c'è un'unica classe). Avendo specificato un modulo come primo parametro, il risultato della compilazione sarà posizionato all'interno del modulo (quindi nella cartella mods/cdsc.mymodule). Come ultimo parametro chiediamo al compilatore di compilare il descrittore che è sotto la cartella src/cdsc.mymodule. Anche il

risultato della compilazione è visibile nella figura 19.1. Adesso possiamo eseguire la nostra classe con il seguente comando:

```
java --module-path mods -m
cdsc.mymodule/java9.cap19.HelloModularWorld
```

che stamperà:

```
Hello Modular World!
```

Infatti, con l'opzione --module-path, abbiamo specificato il cosiddetto **modulepath**, ovvero il percorso dove trovare il modulo da eseguire, che nel nostro caso coincide con la cartella mods. È possibile anche specificare più cartelle (contenenti moduli) separati dal simbolo;.

> **Il modulepath dovrebbe mandare in pensione la storica variabile `classpath` che è stata estensivamente utilizzata sino a Java 8 (cfr. appendice E).**

L'opzione **-m** serve per specificare al comando **java** che stiamo eseguendo un file che appartiene al modulo principale (il main module) che contiene la classe con il `main()`.

19.2.2 Nozioni architetturali e di progettazione

Prima di continuare a parlare di moduli è opportuno comprendere alcuni concetti e conoscere alcune definizioni sulla progettazione e sull'architettura. Il modulo non è una classe, né un oggetto, né un metodo, non definisce cicli e condizioni, non rappresenta un classico elemento di programmazione. Questo può lasciare spiazzati inizialmente, e potrebbe risultare deleterio accettare la sua introduzione in Java senza comprendere le ragioni che ci sono alla base. Quindi la prima cosa da comprendere è che un modulo non è un elemento di programmazione, bensì uno strumento per gestire il **partizionamento del software.** Partizionare il software è una attività architetturale che permette la suddivisione dell'applicazione complessa in più sottosistemi. Un **sottosistema** è costituito da un insieme di classi relazionate da associazioni,

eventi e vincoli, e per il quale è possibile uno sviluppo indipendente dagli altri sottosistemi. Quando un sottosistema astrae una funzionalità rilevante del sistema si parla di **partizionamento verticale.** Ovvero l'applicazione viene divisa per funzionalità della stessa importanza, e ognuna di esse può essere sviluppata indipendentemente dall'altra. Nella figura 19.2 è rappresentato con un component diagram UML un esempio di partizionamento verticale, dove per una certa applicazione gestionale di vendita, sono estratti tre sottosistemi: la `Vendita` che dipende dal `Magazzino`, e la `Fatturazione` che dipende dalla `Vendita`.

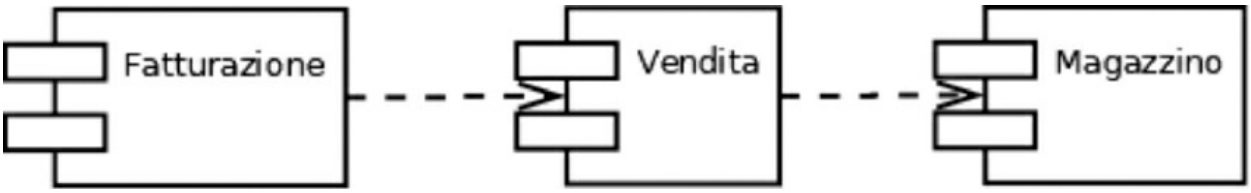


Figura 19.2 - Partizionamento verticale: i sottosistemi di un gestionale.

Questo significa che i diversi sottosistemi possono essere sviluppati in parallelo da persone diverse, coordinate da un responsabile che conosce come procedono gli sviluppi per non farli divergere. Attenzione, esisterà sempre un certo grado di dipendenza tra i vari sottosistemi, altrimenti essi non potrebbero interagire tra loro nella stessa applicazione. Tale grado di dipendenza viene detto **coupling** (in italiano **accoppiamento).** L'accoppiamento è quindi una metrica che misura la dipendenza tra classi, tra package, tra metodi e così via.

Il concetto di **dipendenza** è molto ampio, ma nella maggior parte dei casi implica che del codice non compila senza il codice da cui dipende. Per esempio una classe dipende fortemente da un'altra se ne dichiara una variabile d'istanza, e ancora di più se questa variabile è istanziata esplicitamente con l'operatore `new`. È un po' meno dipendente se ne dichiara un reference per esempio come parametro di un metodo. Se invece viene utilizzato un reference di una sua superclasse invece del reference dell'implementazione della classe stessa, allora l'accoppiamento diminuisce. Questa tecnica è stata consigliata più volte quando abbiamo parlato di collection e generics. In particolare avevamo accennato, nella nota del paragrafo 17.2.2.2 sull'interfaccia `Iterator`, al **principio di inversione della dipendenza.** Noto come **DIP** (acronimo di **Dependency Invertion**

Principle), è uno dei principi fondamentali della progettazione object oriented. Esso afferma che le classi devono dipendere dalle astrazioni e non dalle implementazioni.

Il nostro obiettivo per quanto riguarda i sottosistemi, è di mantenere il più basso possibile l'accoppiamento tra essi, ottenendo quello che viene detto **loose coupling** (che possiamo tradurre come **basso accoppiamento).**

Un'altra misura che viene utilizzata per partizionare il software (e quindi definire sottosistemi) e valutarne la qualità è la **cohesion** (in italiano **coesione).** Essa è la misura di quanto un certo elemento (classe, metodo, package, sottosistema, etc.) contribuisce a realizzare un certo scopo all'interno del sistema. I sottosistemi sono altamente coesi (ovvero dichiara solo tipi che collaborano tra loro per realizzare un determinato scopo) quando internamente sono presenti tipi altamente accoppiati ed esternamente sono bassamente accoppiati con tipi di altre classi.

Esistono anche sottosistemi ottenuti mediante un **partizionamento orizzontale**, che è un partizionamento basato soprattutto sull'ereditarietà. Un tipico esempio è quello di un **framework**, ovvero una micro-architettura che mette a disposizione un modello estendibile per realizzare applicazioni nell'ambito di uno specifico dominio.

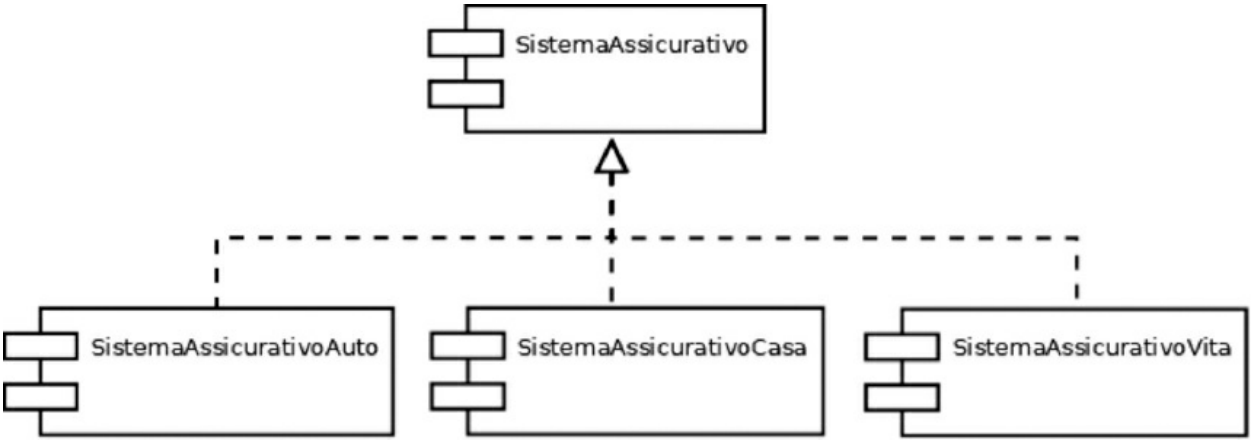


Figura 19.3 - Partizionamento orizzontale: il framework SistemaAssicurativo.

Per esempio nella figura 19.3 è mostrato un esempio (con un component diagram UML) di partizionamento orizzontale dove un framework `SistemaAssicurativo` è implementato concretamente da tre sottosistemi: `SistemaAssicurazioniAuto`, `SistemaAssicurazioniCasa` e

`SistemaAssicurazioniVita`.

Un'ultima definizione che vogliamo dare è quella di **componente software.** Abbiamo definito nel paragrafo 6.2.2 un componente software, come un insieme di classi con un'interfaccia ben definita, che mette a disposizione funzionalità eseguibili indipendentemente dal contesto. Un componente software è quindi un sottosistema eseguibile e riutilizzabile.

Sino ad oggi in Java i sottosistemi come i componenti software e i framework sono stati distribuiti sotto forma di file JAR, o cartelle di file JAR. Con l'avvento di Java 9 cambierà tutto! Infatti il modulo è un perfetto strumento per creare sottosistemi con interfacce ben definite e che godono di un forte incapsulamento, e i JAR modulari sostituiranno i file JAR.

19.2.3 Direttive

Dopo aver visto come si dichiara un modulo, possiamo ora iniziare a parlare più in dettaglio delle direttive. Esse governano essenzialmente i rapporti di **dipendenza tra moduli.** Per quanto detto sul forte incapsulamento, i package contenuti in un modulo sono accessibili ad altri moduli solo se esportati esplicitamente dal modulo in cui risiedono. Inoltre, gli altri moduli devono esplicitamente richiedere tali package al modulo in cui sono definiti. Se un modulo `A` utilizza codice dichiarato in un altro modulo `B`, si dice che `A` **dipende** da `B`. Si dice anche che `A` **legge** `B` e `B` è **letta** da `A`.

Iniziamo ad esplorare le varie direttive partendo dall'unica che abbiamo usato nell'esempio precedente. Infatti anche se non lo abbiamo fatto esplicitamente, ogni modulo dipende sempre dal modulo `java.base`. Anche per i moduli quindi, ci sono comandi che vengono inseriti implicitamente nel nostro codice.

19.2.3.1 `requires`

Ogni descrittore di modulo infatti, dichiara implicitamente la direttiva `requires`. Per esempio il modulo definito nell'esempio precedente dichiarava una direttiva implicita di default:

```
module cdsc.mymodule {
    requires java.base; // direttiva implicita
}
```

È possibile verificarlo eseguendo da riga di comando, sempre dalla cartella radice, il seguente comando che descrive il nostro modulo, specificando la posizione del modulo nella cartella mods tramite il modulepath:

```
java --module-path mods --describe-module cdsc.mymodule
```

Questo comando restituirà l'output:

```
cdsc.mymodule
file:///C:/java9/capitolo_19/hello-modular-world
mods/cdsc.mymodule/
requires java.base mandated
contains java9.cap19
```

dove sono specificati, il nome del modulo seguito dal percorso su disco, la direttiva `requires` implicita (il termine "mandated" può essere interpretato come "obbligatoria" o "imposta") e la lista dei package contenuti, che nel nostro caso contiene un unico package.

La direttiva `requires` implica che il modulo `cdsc.mymodule` dipende dal modulo `java.base`, ovvero che ha accesso ai package che vengono *esportati* da `java.base` (cfr. paragrafo 19.2.2.2).

Esiste anche la possibilità di utilizzare la direttiva `requires transitive`. Quest'ultima garantisce la cosiddetta **implied readability** (in italiano **leggibilità implicita).** Senza di essa, se il modulo `A` dipende dal modulo `B`, e il modulo `B` dipende dal modulo `C`, `A` non dipende da `C` automaticamente. Ma ci sono casi in cui utilizzare `C`, senza poter utilizzare `B`, risulta innaturale (per esempio se `B` dichiarasse qualche tipo che estende qualche tipo dichiarato in `C`). In questi casi è possibile che `B` dichiari la sua dipendenza nel descrittore del modulo nel modo seguente:

```
module B {
    requires transitive C;
}
```

A quel punto, se `A` legge `B`, `A` potrà leggere transitivamente anche `C`. Per esempio il modulo `java.se` (dove "se" sta per "standard edition") non contiene package, ma dichiara transitivamente tutti i package della piattaforma. Questo significa che dipendere dal modulo `java.se`, permette di leggere tutti i package della piattaforma Java. Ovvero si ritorna alla

situazione pre-Java 9 dove avevamo a disposizione il file rt.jar (il che non è consigliabile).

Otteniamo in compilazione un errore di **dipendenza ciclica** (in inglese **cyclic dependance)** se un modulo richiede sé stesso:

```
module ModuloUno {
    requires ModuloUno;
}
```

oppure se più moduli si richiedono a vicenda formando una dipendenza ciclica. Per esempio il modulo `ModuloUno` richiede il modulo `ModuloDue`, il modulo `ModuloDue` richiede il modulo `ModuloUno`. Ma anche se `ModuloUno` richiede `ModuloDue`, `ModuloDue` richiede `ModuloTre` e `ModuloTre` richiede `ModuloUno` e così via. Nella figura 19.4 sono riportati graficamente questi tre esempi con un component diagram. È facile intuire perché questo tipo di dipendenza si chiami ciclica.

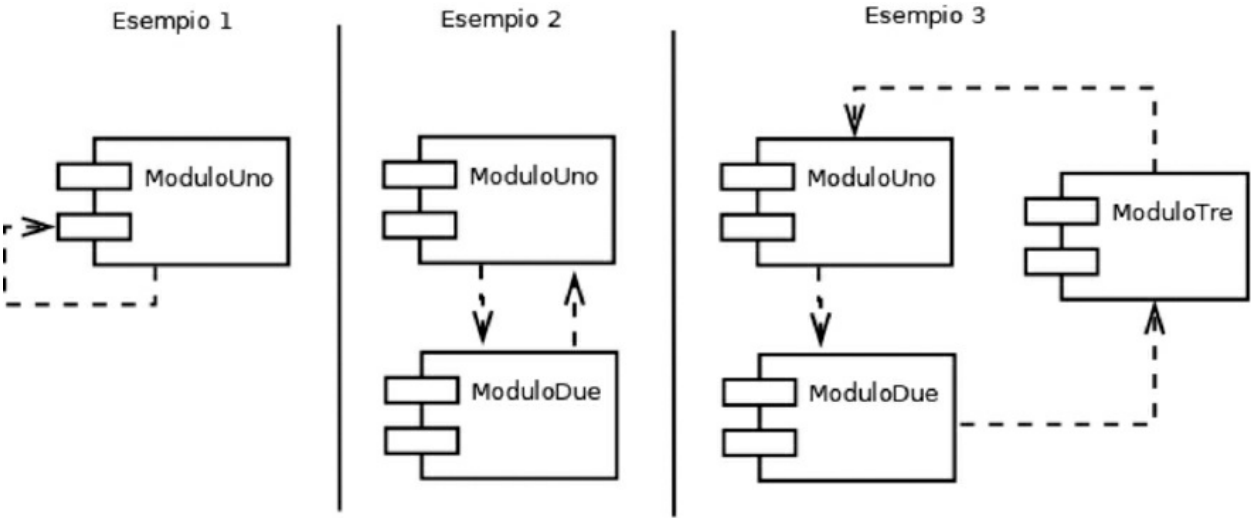


Figura 19.4 - Esempi di dipendenza ciclica.

19.2.3.2 `exports`

La direttiva `exports` consente a un modulo di specificare quali package rendere disponibili alla lettura ad altri moduli che dipendono da esso. Per esempio diamo uno sguardo alla definizione del modulo `java.base`. Potremmo anche aprire il suo descrittore, posizionato all'interno del file src.zip che si trova nella cartella lib della directory di installazione del JDK.

Ma si fa prima ad eseguire il seguente comando dalla prompt di DOS:

```
java --describe-module java.base
```

per ottenere la (lunga) lista delle direttive del modulo `java.base`. Dall'output possiamo estrarre una lista di diversi package che abbiamo studiato essere esportati dal modulo:

```
exports java.io
exports java.lang
exports java.lang.annotation
exports java.lang.reflect
exports java.math
exports java.net
exports java.nio
exports java.nio.file
exports java.time
exports java.time.chrono
exports java.time.format
exports java.time.temporal
exports java.time.zone
exports java.util
exports java.util.concurrent
exports java.util.concurrent.atomic
exports java.util.concurrent.locks
exports java.util.stream
```

Quindi il modulo `cdsc.mymodule` potrà utilizzare tutti questi package (e tanti altri). Con la direttiva `exports to` è possibile esplicitare una lista esclusiva di moduli per i quali verrà esportato il package specificato. In questo caso si parla di **qualified export** (in italiano **esportazione qualificata).** Per esempio `java.base` usa spesso questa direttiva per esportare librerie non standard ad altri moduli, come per esempio:

```
  exports sun.nio.cs to java.desktop, jdk.charsets;
```

In questo caso il package `sun.nio.cs` non potrà essere utilizzato da `cdsc.mymodule`, bensì solo dai package `java.desktop` e `jdk.charsets`.

19.2.3.3 opens *e* open *(forte incapsulamento)*

Una delle caratteristiche più importanti dei moduli è il **forte**

incapsulamento. Prima di Java 9 l'incapsulamento delle classi poteva essere facilmente violato con la reflection (cfr. capitolo 12). Infatti dopo aver invocato il metodo `setAccessible(true)` che si poteva invocare su oggetti tipo `Method` e `Field`, era possibile accedere anche ai membri privati di una classe. Per esempio, consideriamo la seguente classe che dichiara due membri privati:

```
public class OggettoIncapsulato {
    private int x;
    private void metodoPrivato() {
        System.out.println("Questo è un metodo privato");
    }
}
```

Il seguente codice:

```
OggettoIncapsulato o = new OggettoIncapsulato();
Class<?> oClass = o.getClass();
Field privateField = oClass.getDeclaredField("x");
privateField.setAccessible(true);
System.out.println(privateField.get(o));
privateField.set(o, 2);
System.out.println(privateField.get(o));
Method privateMethod =
oClass.getDeclaredMethod("metodoPrivato");
privateMethod.setAccessible(true);
privateMethod.invoke(o);
```

produrrà il seguente output:

```
0
2
Questo è un metodo privato
```

Infatti il metodo `setAccessible()` permette di rendere accessibile alla lettura e alla scrittura gli attributi privati della classe, nonché di invocare metodi privati. Quindi, nel momento in cui una classe è pubblica, anche i suoi membri privati possono essere accessibili!

Con il forte incapsulamento dei moduli, è impedita di default la violazione tramite reflection dell'incapsulamento. Ovvero, se nella classe `HelloModularWorld` aggiungessimo le seguenti linee di codice:

```
 Class<?> oClass = Throwable.class;
 Field privateField =
 oClass.getDeclaredField("detailMessage");
 privateField.setAccessible(true);
```

otterremo il seguente errore al runtime:

```
Exception in thread "main"
  java.lang.reflect.InaccessibleObjectException: Unable to
make field
  private java.lang.String java.lang.Throwable.detailMessage
  accessible: module java.base does not "opens java.lang" to
module
  cdsc.mymodule
```

Qui entrano in gioco le restricted word `open` e `opens`. La direttiva `opens` consente di specificare un package per renderlo disponibile solo a runtime, rendendo quindi accessibile tramite reflection tutti i suoi tipi pubblici (e tutti tipi innestati pubblici o protetti). Con “solo a runtime”, intendiamo dire che se apriamo ma non esportiamo un package, non è possibile utilizzare i suoi tipi pubblici in fase di compilazione. Questo significa che non possiamo importare i tipi pubblici appartenenti al package aperto. Potremmo farlo solo nel caso il package in questione fosse anche esportato. Possiamo però utilizzare tali tipi al runtime, se nel nostro codice abbiamo sfruttato la reflection. Infatti essa ci permette di utilizzare una certa classe senza mai importarla: sfruttando il metodo `forName()` della classe `Class` che prende in input una stringa con un identificatore:

```
 Class<?> oClass = Class.forName("OggettoIncapsulato");
```

Se avessimo anche esportato il package (è legale esportare e contemporaneamente aprire un package) allora avremmo anche potuto importare direttamente il tipo `OggettoIncapsulato`:

```
 OggettoIncapsulato o = new OggettoIncapsulato();
 Class<?> oClass = o.getClass();
```

La sintassi della direttiva per aprire un package al runtime è la seguente:

```
 opens nomePackage;
```

oppure per aprire questo package solo a particolari moduli è possibile

realizzare un**'apertura qualificata** (in inglese **qualified export**):

```
  opens nomePackage to cdsc.mymodule;
```

Invece, se si vogliono aprire tutti i package contenuti nel modulo, si può usare la restricted word `open` accanto alla dichiarazione del modulo:

```
  open module nomeModulo { //...
```

Aprire un modulo completamente è un'operazione sconsigliabile. Si dovrebbe fare solo quando non si è sicuri su quali tipi è utilizzata la reflection al runtime da una certa libreria o framework. Il perché sia così importante l'introduzione delle aperture nei moduli infatti, dipende proprio dal fatto che molti framework come Spring e Hibernate non funzionerebbero più con le nuove regole introdotte dai moduli. Infatti tali framework usano estensivamente la reflection. Più precisamente a volte è necessario aprire parti private del JDK a determinati framework. Purtroppo in questi casi non è possibile ricompilare un modulo del JDK aggiungendo nel descrittore l'apertura con `opens`. Per questi casi è stato introdotto l'opzione del comando java:

```
-add-opens nomeModulo/packageDaAprire=nomeModuloClient
```

L'utilizzo di questa opzione è equivalente ad aprire in maniera qualificata il `packageDaAprire` al modulo che la deve utilizzare, ovvero è equivalente a questa direttiva:

```
  open module packageDaAprire to nomeModuloClient
```

Con la differenza di essere specificata solo a runtime. Per esempio per aprire ad un modulo denominato `moduloframework` il codice privato dei tipi del package `java.lang`, è possibile utilizzare questa sintassi:

```
-add-opens java.base/java.lang=moduloframework
```

Questa opzione va considerata come una scappatoia, e deve essere utilizzata solo se non esistono altre alternative.

19.2.3.4 `uses` e `provides to`

Le direttive `uses` e `provide to` consentono di implementare i cosiddetti servizi con le API di `ServiceLoader` (argomento del prossimo paragrafo). La direttiva `provides to`, segue la seguente sintassi:

```
  provides service_provider_interface with implementazione;
```

dove `service_provider_interface` è un'interfaccia (o classe astratta o classe concreta) implementata da `implementazione`. Con questa direttiva dichiariamo ad eventuali `ServiceLoader` che possono leggere il modulo corrente, che se vogliono utilizzare un'istanza di `service_provider_interface` gli sarà restituito un oggetto di tipo `implementazione`.

Un modulo che dichiara un `ServiceLoader` e vuole utilizzare un'istanza di `service_provider_interface` deve dichiarare nel suo descrittore la direttiva `uses` con la seguente sintassi:

```
  uses service_provider_interface;
```

Rimandiamo il lettore allo studio del prossimo paragrafo per comprendere i dettagli di utilizzo di entrambe le direttive.

19.3 Servizi con ServiceLoader

La classe `ServiceLoader` esiste già dalla versione 6 di Java, e nonostante molti framework importanti basino le loro implementazioni proprio sul concetto di *servizio*, questa caratteristica rimane ai più sconosciuta. La classe `ServiceLoader` è stata rivista per essere utilizzata nel nuovo contesto di modularità. Di seguito creeremo un esempio di applicazione modulare un po' più complesso, prima sfruttando solo il pattern factory, e poi evolvendolo implementando un servizio con `ServiceLoader` che eliminerà l'accoppiamento tra due moduli.

19.3.1 Pattern Factory

Come abbiamo detto un modulo rappresenta il miglior modo per creare sottosistemi. In un'applicazione i vari sottosistemi dovrebbero essere caratterizzati da un basso accoppiamento. Esistono varie strategie per ridurre

l'accoppiamento tra le classi. Uno dei più famosi è quello di utilizzare il **Factory Method Pattern**, che viene spesso chiamato semplicemente **Factory Pattern.** Si tratta di un pattern creazionale GoF (cfr. appendice D) ed è uno dei pattern più utilizzati in assoluto. È basato sul principio di inversione della dipendenza e la strategia d'implementazione è molto semplice. Supponiamo di avere il seguente problema: una certa classe (che chiameremo **classe consumer)** deve compiere una certa azione su oggetti diversi tra loro ma con un'interfaccia comune. Se la nostra classe dipendesse da questi oggetti allora la manutenzione sarebbe complicata. Infatti se volessimo aggiungere un nuovo oggetto da gestire, deve cambiare l'implementazione anche della classe consumer. Questo perché la classe consumer ha un alto accoppiamento con i vari oggetti. Per ridurre la dipendenza tra la classe consumer e gli oggetti stessi, si può far dipendere la classe consumer solo dall'interfaccia e non dalle implementazioni (applicazione del principio DIP). Inoltre si può creare una classe (che chiameremo **factory)** che si occuperà di istanziare gli oggetti, disaccoppiando di fatto la classe consumer dagli oggetti.

19.3.2 Factory modulare

Per esempio supponiamo di avere una classe che deve aggiungere un trademark (un marchio di fabbrica) a diverse tipologie di documenti. Potremmo implementare il pattern factory creando un'interfaccia `Document` che dichiara il metodo `addTrademark()` e viene implementata dalle sottoclassi `Certificate` (in italiano "certificato") e `Invoice` (in italiano "fattura").

> **Nel nostro esempio `Document` è un'interfaccia, ma tecnicamente potrebbe essere anche una classe astratta, o addirittura una classe concreta.**

Tali sottoclassi implementano concretamente il metodo `addTrademark()` nella maniera corretta per il tipo di documento. Poi una classe `DocumentFactory` si occuperà di definire un metodo factory che restituisce il

tipo di documento:

```
public Document getDocument(String documentId) {
    Document document;
    switch (documentId) {
        case "I":
            document = new Invoice();
            break;
       case "C":
           document = new Certificate();
           break;
       default:
           document = null
           break;
    }
    return document;
}
```

Con questa soluzione è vero che abbiamo eliminato l'accoppiamento tra la classe consumer e gli oggetti concreti, ma la classe `DocumentFactory` ha ereditato un alto accoppiamento. Questo è accettabilissimo quando abbiamo a che fare con le classi, ma con l'avvento della modularità in Java 9, dobbiamo tener conto del forte incapsulamento. In figura 19.5 è rappresentato un component diagram che mostra l'architettura modulare scelta per questa applicazione.

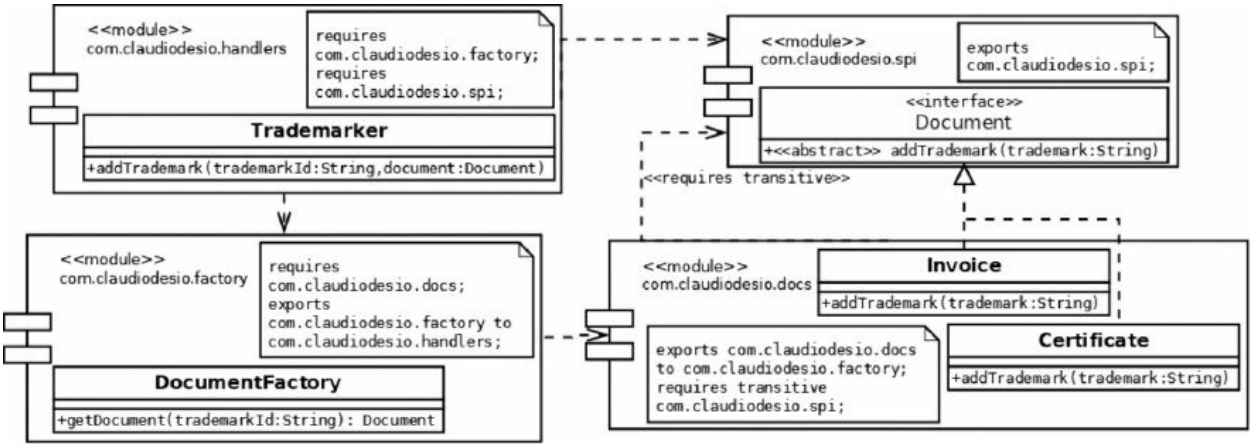


Figura 19.5 - Component diagram che mostra l'architettura modulare dell'applicazione.

La notazione di componente è stata utilizzata per definire quattro moduli. Lo si può capire chiaramente notando lo stereotipo `<<module>>` con cui sono stati

marcati. Ognuno di essi contiene la notazione di classe (o di interfaccia) che definisce, e una nota (foglio con angolo piegato) che mostra le direttive dichiarate nel descrittore del modulo. Inoltre sono indicate anche le dipendenze tra i moduli, che derivano dalle direttive di tipo `requires`.

In questo esempio abbiamo utilizzato le convenzioni utilizzate da Oracle per i nomi dei moduli (che coincidono con quelle dei package). Tali identificatori utilizzano infatti la cosiddetta **reverse domain name convention** (in italiano **convenzione del nome del dominio rovesciato).** Laddove applicabile, essa consiglia di utilizzare come prefisso dell'identificatore il nome del dominio aziendale rovesciato. Quindi il dominio "claudiodesio.com", si rovescia e diventa il prefisso di tutti i moduli che abbiamo definito: `com.claudiodesio`. Poi il suffisso (che può anche essere composto da più stringhe separate sempre da punti) identificherà univocamente il modulo. Tutte le lettere devono essere minuscole. Siccome anche i package seguono la stessa convenzione, in questo esempio abbiamo una corrispondenza uno a uno tra i nomi dei moduli e i nomi dei package contenuti (anche questa corrispondenza è una convenzione consigliata).

Si noti come il modulo `com.claudiodesio.spi`, contenga solo l'interfaccia `Document` che viene esportata. Il suffisso `spi` è l'acronimo di **service provider interface** (in italiano **interfaccia fornitrice del servizio).** Infatti vedremo che rappresenterà la pietra angolare del concetto di *servizio*. Questo modulo non dipende da nessun altro.

Il secondo modulo `com.claudiodesio.docs`, contiene le implementazioni dell'interfaccia `Document: Invoice` e `Certificate`. Il package contenente queste due classi viene esportato solo per il modulo `com.claudiodesio.factory`. Questo modulo inoltre dipende dal modulo `com.claudiodesio.spi` transitivamente. In effetti sembra una scelta sensata permettere a chi usa le due implementazioni di poter usare anche l'interfaccia da loro implementata.

Quindi, dato che il modulo `com.claudiodesio.factory` dipende da `com.claudiodesio.docs`, per la proprietà transitiva dipende anche da `com.claudiodesio.spi`.

Inoltre espone il package `com.claudiodesio.factory` (che contiene la classe `DocumentFactory`) solo al modulo `com.claudiodesio.handlers`.

Il modulo `com.claudiodesio.handlers` contiene la classe `Trademarker`,

che definisce il metodo `addTrademark()`; quest'ultimo usa `DocumentFactory` per ottenere un'istanza di `Document`:

```
public void addTrademark(String documentld, String
trademark) {
  Document document =
DocumentFactory.getDocument(documentId);
  document.addTrademark(trademark);
}
```

Ma `DocumentFactory` non importa le implementazioni concrete (e quindi è indipendente da esse). Infatti legge solo i moduli `com.claudiodesio.spi` e `com.claudiodesio.factory`. Sinoti che aquesta applicazione manca una classe con il `main()` per essere eseguita. A tale proposito abbiamo implementato un modulo aparte (non riportato nel precedente programma) chiamato `com.claudiodesio.client`, contenente la classe eseguibile chiamata `TrademarkClient`, che implementa una sessione interattiva per richiamare il metodo `addTrademark()`. Questo modulo legge il modulo `com.claudiodesio.handlers`. Come sempre tutto il codice riportato su questo libro è disponibile on line, con i lunghi comandi di compilazione ed esecuzione già pronti.

L'applicazione funziona, ma ha un difetto per quanto riguarda l'architettura modulare: l'esposizione delle classi `Invoice` e `Certificate` al modulo `com.claudiodesio.factory`. Infatti, supponiamo che con l'evoluzione del software ci sia bisogno di introdurre un nuovo tipo di documento, per esempio la classe `CreditNote` che rappresenta una nota di credito. Probabile che questa sia inclusa nel modulo `com.claudiodesio.docs` (anche se potrebbero esserci diverse ragioni per creare un modulo a parte) con lo scopo di incapsularla, senza che il resto dell'applicazione cambi.

Purtroppo possiamo anche compilare un nuova versione di questo modulo, ma se non aggiorniamo anche la classe `DocumentFactory` del modulo `com.claudiodesio.factory`, allora `CreditNote` non sarà mai utilizzata. Per il forte incapsulamento, l'ideale sarebbe che l'aggiunta della classe `CreditNote`, non comporti la modifica di altri moduli. Questo è possibile (in maniera standard) tramite appunto il concetto di **servizio** e la classe `ServiceLoader`.

19.3.3 Evoluzione con ServiceLoader

Con la classe `ServiceLoader`, possiamo ora eliminare completamente la dipendenza tra moduli (e quindi moltiplicare la possibilità di riutilizzo del modulo stesso in altre applicazioni) ed implementare il vero forte incapsulamento. Per implementare questa soluzione dobbiamo rivedere l'architettura modulare della nostra applicazione, e fare delle piccole modifiche al codice. Nella figura 19.6 è rappresentata la nuova architettura modulare dell'applicazione.

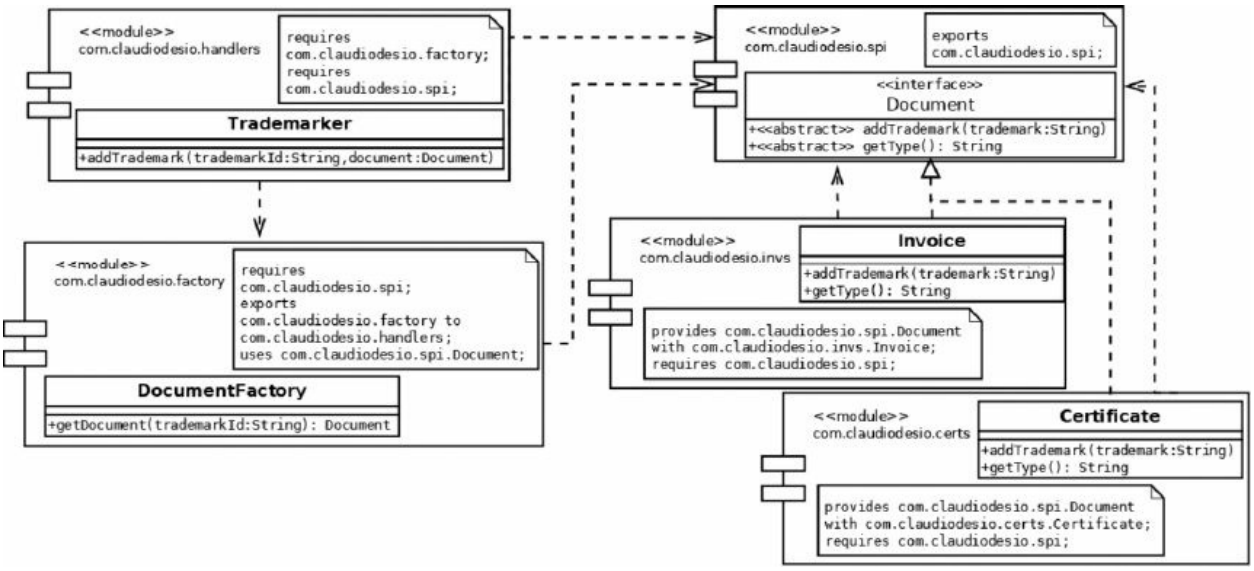


Figura 19.6 - Architettura modulare rivista per implementazione servizio.

Si noti che è stato eliminato il modulo `com.claudiodesio.docs` e le implementazioni di `Document` sono state spostate in moduli separati: `com.claudiodesio.certs` per la classe `Certificate`, e `com.claudiodesio.invs` per la classe `Invoice`. Entrambi i moduli non esportano tali implementazioni, realizzando quindi un forte incapsulamento. Per entrambe le implementazioni, i rispettivi moduli utilizzano invece la direttiva `provides` mediante la seguente sintassi:

```
provides service_provider_interface with implementazione;
```

Dove `service_provider_interface` è l'interfaccia (o classe astratta o classe concreta) implementata da `implementazione`. Nella nostra applicazione il descrittore del modulo `com.claudiodesio.certs` dichiara che fornirà

un’istanza di `Certificate` al `ServiceLoader` che cercherà di caricare implementazioni di `Document` con la seguente direttiva:

```
  provides com.claudiodesio.spi.Document with
    com.claudiodesio.certs.Certificate;
```

Mentre il descrittore del modulo `com.claudiodesio.inves` dichiara che fornirà al `ServiceLoader` un’istanza di `Invoice` come implementazione di `Document`:

```
  provides com.claudiodesio.spi.Document with
    com.claudiodesio.invs.Invoice;
```

Invece il modulo `com.claudiodesio.factory` dichiara la direttiva `uses` con la seguente sintassi:

```
  uses service_provider_interface;
```

In particolare dichiara `Document` come `service_provider_interface`:

```
  uses com.claudiodesio.spi.Document;
```

Questo implica che un’eventuale `ServiceLoader` dichiarato all’interno del modulo, proverà a caricare implementazioni di `Document` dagli altri moduli. Infatti nella classe `DocumentFactory`, il metodo `getDocument()` è stato modificato per utilizzare un oggetto `ServiceLoader`:

```
  public static Document getDocument(String documentType) {
      ServiceLoader<Document> serviceLoader =
        ServiceLoader.load(com.claudiodesio.spi.Document.class);

          for (Document document :  serviceLoader) {
              if (document.getType().equals(documentType)) {
                  return document;
              }
          }
      }
      throw new IllegalArgumentException(
        "Nessun documento trovato con ID=" + documentId);
  }
```

L’utilizzo di `ServiceLoader` è davvero semplice. Il metodo `load()`

restituisce un oggetto `ServiceLoader` carico di tutte le istanze delle implementazioni che i moduli disponibili forniscono. Nel nostro caso vengono istanziate al volo (e senza problemi di performance) `Certificate` e `Invoice`. La classe `ServiceLoader` implementa `Iterable`, la qual cosa consente di eseguire cicli facilmente su esso mediante un ciclo for migliorato, restituendo l'istanza corrispondente al parametro `documentType` (questo sarà specificato dall'utente dell'applicazione). Dalla figura 19.6 infatti, si può notare che nell'interfaccia `Document` è stato aggiunto il metodo `getType()`, che viene implementato nelle sottoclassi (`Certificate` restituisce la stringa `C`, e `Invoice` la stringa `I`).

La cosa più importante in questa classe, è che è assolutamente indipendente da `Certificate` e `Invoice`, che come abbiamo visto non sono neanche esportate dai rispettivi moduli.

19.3.4 Compilazione ed esecuzione

Si consiglia di compilare ed eseguire l'applicazione (sia senza implementazione di servizio, sia con implementazione del servizio) usando il codice disponibile on line (e che probabilmente avete già scaricato). In questo caso abbiamo anche fornito degli script da eseguire, visto che gli script di compilazione sono davvero verbosi. Inoltre troverete anche uno script di compilazione dove intenzionalmente non viene compilato il modulo `com.claudiodesio.invs`. In questo caso nel momento in cui l'utente prova a richiedere un documento di tipo `Invoice`, il flusso del metodo `getDocument()` farà scattare la `IllegalArgumentException` al runtime. Infatti il `ServiceLoader` caricherà solamente un'istanza di `Certificate` visto che non troverà il modulo di `Invoice`. Se aggiungessimo altri moduli con altre implementazioni come `CreditNote`, tutti gli altri moduli non dovranno essere modificati, e questo rappresenta un grande vantaggio.

19.3.5 Metodo `provider`

Come vengono istanziate le implementazioni? Visto che non siamo noi a istanziare gli oggetti, è necessario che abbiano un **costruttore senza parametri.** Nel nostro caso, non avendo dichiarato costruttori nelle classi `Certificate` e `Invoice`, possiamo sfruttare il loro costruttore di default che

come sappiamo non ha parametri. Questo però non è sempre desiderabile, spesso si vuole istanziare un oggetto passando parametri al costruttore. Per fare questo possiamo definire il metodo statico senza parametri `provider()` in una classe che istanzia un'implementazione. Per esempio nella stessa classe `Certificate` potremmo inserire un costruttore che prende in input una stringa e aggiunge il seguente metodo `provider()`:

```
public static Document provider() {
    return new Certificate("Creato con provider()!");
}
```

E tutto funzionerebbe come prima. Una soluzione ancora migliore consiste nel creare nello stesso package di `Certificate`, una factory specifica che definisce il metodo `provider()`:

```
public class CertFactory {
    public static Document provider() {
        return new Certificate("Creato con provider()!");
    }
}
```

In tal caso bisogna cambiare la direttiva per specificare questa classe come *provider*:

```
provides com.claudiodesio.spi.Document with
com.claudiodesio.certs.CertFactory;
```

Ora possiamo addirittura eliminare il modificatore public dalla definizione della classe `Certificate`, realizzando un incapsulamento davvero forte, visto che questa classe non sarà visibile al di fuori del package!

19.4 Jar modulare

Come descritto nell'appendice E, un **file JAR** è un file che serve per impacchettare le nostre classi, e da sempre è il modo preferito per distribuire codice Java. Il termine **JAR** sta per **Java ARchive** (in italiano **archivio java)** e non è altro che un file compresso dal comando jar del JDK. Il formato binario di un file JAR è identico a quello del formato ZIP, e contiene le classi Java così come sono state compilate. Ha come suffisso .jar, e definisce al suo

interno un file descrittore che si chiama manifest.mf, a cui ci si referenzia come **file manifest.** Questo file deve risiedere all'interno di una cartella chiamata META-INF. All'interno del file manifest si possono scrivere diverse informazioni per descrivere il file JAR. Per esempio si può specificare la variabile classpath (cfr. appendice E), come anche la classe contenente il metodo `main()` con l'istruzione:

```
  Main-Class:classe_con_il_metodo_main
```

che rende di fatto eseguibile il file JAR. Infatti, tramite il comando:

```
java -jar nome_file_jar
```

viene eseguito il metodo `main()` della classe specificata nel manifest del file JAR. Con l'introduzione dei moduli, possiamo utilizzare ancora i file JAR? Sino ad ora il contenuto dei nostri moduli compilati sono stati salvati in una cartella mods sull'hard disk, ma è anche possibile impacchettare un modulo in un cosiddetto file **JAR modulare** (in inglese **modular JAR).** Si tratta solo di un file JAR ordinario, che contiene un descrittore (module-info.class) nella sua directory radice, questa è l'unica differenza strutturale. Il comando jar è stato modificato in Java 9 per supportare la creazione dei file JAR modulari.

19.4.1 Strumento jar

Se vogliamo impacchettare in JAR modulari i moduli creati nell'esempio del paragrafo precedente, dobbiamo utilizzare i comandi:

```
jar --create --file=lib/com.claudiodesio.spi.jar
  --module-version=1.0 -C mods/com.claudiodesio.spi .
jar --create --file=lib/com.claudiodesio.invs.jar
  --module-version=1.0 -C mods/com.claudiodesio.invs .
jar --create --file=lib/com.claudiodesio.certsjar
  --module-version=1.0 -C mods/com.claudiodesio.certs .
jar --create --file=lib/com.claudiodesio.factory.jar
  --module-version=1.0 -C mods/com.claudiodesio.factory .
jar --create --file=lib/com.claudiodesio.handlers.jar
  --module-version=1.0 -C mods/com.claudiodesio.handlers .
jar --create --file=lib/com.claudiodesio.client.jar
  --module-version=1.0 -main-class=
  com.claudiodesio.client.TrademarkClient
  -C mods/com.claudiodesio.client .
```

In questo modo abbiamo impacchettato i sei moduli creati, con sei comandi simili. Solo nell'ultimo abbiamo specificato anche il nome della classe del `main()` tramite l'opzione --main-class, rendendolo un file JAR modulare eseguibile. Si noti come sono sempre specificati il nome del file da creare con l'opzione --file e la versione del modulo con l'opzione --module-version, e con l'opzione -C si indichino i contenuti da inglobare specificando la cartella che li contiene, e i file interni a questa cartella (in questo esempi con il simbolo "." intendiamo che devono essere inglobati tutti i contenuti della cartella specificata). Alla fine i file .jar si troveranno nella cartella lib, e sarà possibile eseguire il file JAR con il comando:

```
java -p lib -m com.claudiodesio.client
```

dove -p è l'abbreviativo di `--module-path`, mentre `-m` è l'abbreviativo di -module (che serve a indicare il nome del modulo da eseguire).

Gli strumenti del JDK con Java 9 sono stati rivisitati pesantemente, e le opzioni da specificare ora hanno una versione lunga ed una breve. L'opzione --help digitata dopo ogni comando (java, javac, jar, **etc.) provocherà la stampa della sinossi del comando, specificando tutte le opzioni disponibili.**

19.4.2 Formato jmod

Se abbiamo bisogno di impacchettare oltre a package e risorse anche librerie native (cfr. appendice N), allora bisogna utilizzare **file JMOD.** Esempi di file JMOD si trovano nello stesso JDK, e contengono proprio i moduli della piattaforma come per esempio `java.base`. Sono tutti posizionati nella cartella jmods del JDK ed hanno suffisso jmod. Per creare un file JMOD bisogna utilizzare il nuovo strumento jmod del JDK. Per esempio, con il seguente comando eseguito dalla cartella radice C:/java9/capitolo_19/hello-modular-world:

```
jmod create -class-path mods/cdsc.mymodule hello.jmod
```

creiamo nella cartella radice il file JMOD dell'esempio creato nel paragrafo 19.2.1.

Anche per `jmod` è possibile utilizzare l'opzione --help per conoscere tutte le opzioni disponibili.

19.5 Migrazione di programmi pre-Java 9

Una caratteristica che ha da sempre caratterizzato Java, è la retro-compatibilità con il codice scritto con versioni precedenti. Per oltre vent'anni abbiamo letto nella documentazione che ci sono librerie deprecate che un giorno potrebbero non essere più supportate. Intanto le librerie destinate ad uso interno dal JDK, sono state largamente utilizzate perché comunque accessibili. La retro-compatibilità ha permesso solitamente una migrazione alla versione successiva di Java, quasi sempre indolore. Nella maggior parte dei casi il nostro codice non si è dovuto cambiarlo di una virgola. Invece, applicazioni scritte con versioni precedenti di Java, potrebbero non funzionare se eseguite con Java 9. Questo perché la modularizzazione con il forte incapsulamento ha fatto sì che alcune librerie destinate ad uso interno, che prima di java 9 i programmatori hanno utilizzato, non siano più disponibili perché non esportate dai moduli che le contengono. In questi casi quei programmi non funzioneranno se non si apporteranno delle modifiche, per esempio utilizzando librerie sostitutive. Però il modulo `jdk.unsupported` esporta alcune di queste librerie, quelle senza le quali alcuni framework e tool molto importanti non potrebbero più funzionare. Quindi se proprio non ci sono alternative, è possibile utilizzare questo modulo. Per supportare la migrazione del codice pre-Java 9, il JDK ha rivisto anche lo strumento **jdeps**. Questo è stato introdotto con la versione 8, ed ora è stato modificato per scoprire le dipendenze di un tipo (o di tutti i tipi in un file JAR) da librerie che ora sono fortemente incapsulate. La sintassi di jdeps è la seguente:

```
jdeps --jdk-internals nome_file
```

dove `nome_file` può essere un nome di un tipo, di una directory o di un file JAR.

Anche per jdeps è possibile utilizzare l'opzione --help per conoscere tutte le opzioni disponibili.

19.5.1 Modulo anonimo

Per quanto detto con Java 9 tutti i package dovrebbero appartenere a moduli. Per la filosofia di retro-compatibilità che da sempre contraddistingue Java, è stato implementato un meccanismo che considera tutto il codice che viene puntato dalla variabile classpath e che non appartiene ad un modulo, come appartenente ad un **modulo anonimo** (in inglese **unnamed module).** In questo modo possiamo utilizzare codice non modularizzato con il JDK 9. Così però non potremo usufruire dei vantaggi della modularizzazione. Un modulo anonimo infatti legge tutti gli altri moduli come se avesse dichiarato nel suo descrittore una direttiva `requires` per ogni altro modulo. Inoltre, non avendo un nome, un modulo anonimo non può essere nominato in una direttiva `requires` da altri moduli. Questo implica che un modulo anonimo esporta implicitamente tutti i suoi package. Concludendo, il modulo anonimo rappresenta solo la ragione tecnica per cui possiamo ancora utilizzare codice scritto prima dell'avvento di Java 9.

19.5.2 Modulo automatico

Ereditiamo dalle versioni precedenti di Java, milioni di file JAR che non sono modularizzati. Java 9 consente di rendere tali JAR dei **moduli automatici** (in inglese **automatic modules).** Affinché ciò accada, basta che essi siano puntati dalla variabile modulepath al runtime. A differenza di un modulo anonimo, un modulo automatico ha un nome che coincide con il nome del file JAR, senza il suffisso. Per esempio se il file pos.jar viene puntato dal modulepath al runtime, esso diviene un modulo automatico di nome pos. Un modulo automatico, come un modulo anonimo, esporta implicitamente tutti i suoi package, e legge tutti gli altri moduli dell'applicazione, compresi gli altri moduli automatici. Quindi accede a tutti i tipi pubblici degli altri moduli.

19.6 Jlink

Il JDK, nella versione 9.0.1, definisce 98 moduli. Questo numero è destinato a cambiare (nella versione 9-ea per esempio si contavano 95 moduli). È possibile verificarlo eseguendo il comando:

```
java --list-modules
```

Se invece ci spostiamo nella cartella bin del JRE ed eseguiamo il medesimo comando, scopriremo che vengono definiti solo 76 moduli. Quindi con i moduli è possibile creare diversi runtime. Nei semplici esempi che abbiamo riportato tra le pagine di questo libro, non abbiamo mai usato librerie come `java.sql` per accedere a database (lo faremo nell'appendice R) e `javafx.*` per creare interfacce grafiche (che introdurremo invece nell'appendice S). Eppure, prima di Java 9, queste librerie erano obbligatoriamente inglobate nel runtime Java, anche se non si utilizzavano. Con Java 9, distribuire i nostri programmi portandoci dietro librerie inutilizzate, è assolutamente evitabile. In questo modo si può ridurre la dimensione dell'applicazione, e favorire le performance della stessa.

Java 9 infatti ha introdotto un nuovo strumento chiamato jlink. Esso ci permette di creare dei runtime personalizzati solo con i moduli che servono alla nostra applicazione. Per esempio con il seguente comando creiamo un runtime contenente solo il modulo `java.base`:

```
jlink --module-path "C:/Program Files/Java/jdk-9.0.1/jmods"
  --add-modules java.base --output javabasert
```

Infatti con l'opzione `--module-path` specifichiamo il modulepath alla cartella jmods contenente i moduli del JDK. Con l'opzione `--add-modules` aggiungiamo il modulo `java.base` al runtime. Infine con l'opzione `--output` diamo il nome al runtime (`javabasert`). A questo punto jlink creerà un runtime all'interno di una cartella che si chiama proprio `javabasert`, con tutta la struttura costituita dalle cartelle bin, lib, include, conf, legals. Se eseguiamo il comando:

```
javabasert\bin\java --list-modules
```

otteniamo il seguente output:

```
java.base@9.0.1
```

Se volessimo eseguire il modulo `cdsc.mymodule` che abbiamo creato

all’inizio del capitolo, sfruttando il runtime appena creato, dalla cartella radice C:/java9/capitolo_19/hello-modular-world basterà eseguire il comando:

```
javabasert\bin\java --module-path mods
  -m cdsc.mymodule/java9.cap19.HelloModularWorld
```

Se vogliamo creare un runtime `javabasedeskrt` che includa anche il modulo `java.desktop`, possiamo eseguire il comando:

```
jlink --module-path "C:/Program Files/Java/jdk-9.0.1/jmods"
  --add-modules java.base,java.desktop --output javabasedeskrt
```

e per verificare la corretta composizione dell’ambiente eseguiamo:

```
javabasedeskrt\bin\java --list-modules
```

che produrrà il seguente output:

```
java.base@9.0.1
java.datatransfer@9.0.1
java.desktop@9.0.1
java.prefs@9.0.1
java.xml@9.0.1
```

Riepilogo

Il **Java Platform Module System** porta con sé una **configurabilità affidabile**, che ci permette di gestire problemi di dipendenza già in fase di compilazione, un **forte incapsulamento** per rendere davvero inaccessibili i tipi incapsulati, una **piattaforma Java scalabile** per creare runtime personalizzabili, **maggiore integrità della piattaforma** grazie al forte incapsulamento sulle librerie più vulnerabili (che potrebbe rendere la migrazione di alcuni programmi scritti prima di Java 9 difficoltosa) e **prestazioni migliori.** Il modo in cui dobbiamo pensare allo sviluppo ormai deve cambiare per supportare i vantaggi offerti da un’architettura modulare.

Un **modulo** è una cartella che contiene un gruppo di package relazionati tra loro, opzionalmente altre risorse come immagini e file di configurazione, e un **descrittore del modulo.** In un descrittore del modulo è possibile utilizzare 5 tipi di direttive diverse.

La direttiva **`requires`** consente di specificare il nome di un altro modulo da cui leggere eventuali tipi dei package esportati. A sua volta la direttiva `exports`, permette ad un modulo di specificare quali package rendere disponibili in lettura ad altri moduli. È possibile esportare package anche a specifici moduli con la direttiva **`exports to`.** In tal caso si parla di **esportazione qualificata.** Visto che i moduli supportano il **forte incapsulamento** che non consente di utilizzare la reflection per leggere i membri privati di un tipo, è stata creata la direttiva **`opens`** che permette la lettura al runtime tramite reflection dei package specificati. Stesso discorso per la restricted word **`open`** che viene usata come un modificatore per il modulo, e apre tutti i package dello stesso. La direttiva **`provides to`** specifica quale implementazione di un precisato tipo fornirà ad un eventuale `ServiceLoader`. A sua volta un **`ServiceLoader`** che vuole leggere implementazioni di un certo tipo, deve essere incluso in un modulo che dichiara una clausola **`uses`** che specifica quel tipo. Un metodo **`provider()`** può essere utilizzato per superare il limite di dover utilizzare obbligatoriamente un costruttore senza parametri per istanziare un'implementazione che può leggere il **`ServiceLoader.`** La realizzazione di un **servizio** tramite le API del `ServiceLoader` consente di disaccoppiare completamente moduli che riescono comunque ad interagire.

L**'accoppiamento** è una metrica che misura la dipendenza tra elementi del software. La **coesione** è una metrica che misura quanto un certo elemento contribuisce a realizzare un certo scopo all'interno del sistema.

I moduli sono perfetti per rappresentare **sottosistemi:** questi possiamo definirli come insiemi di classi relazionate da associazioni, eventi e vincoli, e per il quale è possibile uno sviluppo indipendente dagli altri sottosistemi. Essi sono altamente coesi, internamente sono presenti tipi altamente accoppiati, ed esternamente sono scarsamente accoppiati con tipi di altre classi.

Partizionare il software è una attività architetturale, che permette la suddivisione dell'applicazione complessa in più sottosistemi. Esistono **partizionamenti verticali** dove l'applicazione viene divisa per funzionalità della stessa importanza, e **partizionamenti orizzontali** basati soprattutto sull'ereditarietà. Il **principio di inversione della dipendenza** afferma che le classi devono dipendere dalle astrazioni e non dalle implementazioni. Un **componente software** è un sottosistema eseguibile e riutilizzabile.

Molti **strumenti del JDK** sono stati modificati (o creati da zero) per supportare il nuovo modello modulare in Java 9. Il comando jar ora consente di creare file **JAR modulari**, ovvero file JAR creati a partire da moduli, e che tecnicamente differiscono da un file JAR ordinario solo perché contengono il file descrittore nella cartella radice. Il comando jmods permette di creare file di tipo **JMOD**, che rispetto ai file modulari solitamente contiene codice nativo. Lo strumento jlink, invece consente di creare in modo semplice ambienti di runtime personalizzati, includendo solo i moduli davvero necessari. Per supportare la **migrazione a Java 9** lo strumento jdeps è stato modificato per scoprire le dipendenze di un tipo (o di tutti i tipi in un file JAR), da librerie che ora sono fortemente incapsulate. Inoltre tutto il codice puntato dalla variabile classpath al runtime è considerato parte di un **modulo anonimo** in modo trasparente, allo scopo di poter essere compatibile con il nuovo sistema modulare. Viene detto **modulo automatico** un file JAR che è puntato dalla variabile modulepath. Il nome di un modulo automatico coincide con il nome del file senza il suffisso .jar. Sia il modulo anonimo che il modulo automatico esportano tutti i propri package e leggono tutti gli altri moduli.

Esercizi e altro materiale didattico sono disponibili all'indirizzo http://www.claudiodesio.com/java9.html.

Obiettivi del capitolo)

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere le ragioni che hanno portato all'introduzione del nuovo sistema modulare di Java 9 (unità 19.1)	☐	
Saper padroneggiare i moduli e le relative direttive (unità 19.2, 19.3)	☐	
Saper implementare servizi basati sui moduli per eliminare l'accoppiamento tra sottosistemi (unità 19.3)	☐	

Saper creare JAR modulari e file JMOD utilizzando gli strumenti del JDK (unità 19.4)	☐	
Comprendere e saper affrontare le difficoltà per migrare applicazioni scritte in passato a Java 9 (unità 19.5)	☐	
Saper creare runtime personalizzati con lo strumento jlink (unità 19.6)	☐	

Note:

Conclusioni

Qui termina “Manuale di Java 9”, anzi no! In realtà, come più volte detto, tra le appendici e gli esercizi, on line trovate tanto materiale che quasi raddoppia il numero di pagine di questo libro. Tra le appendici in particolare, trovate veri e propri capitoli che solo per ragioni di spazio sono stati spostati in rete. Studiando le appendici potrete imparare a connettere le vostre applicazioni alla rete, creare interfacce grafiche con diverse librerie e la tecnologia Java FX, connettervi a database e così via. C’è ancora tanto da apprendere!

Speriamo che questo libro sia stato utile per raggiungere i vostri obiettivi, e vi consigliamo di seguire l’autore on line tramite i seguenti canali:

- Facebook: http://www.facebook.com/claudiodesiocesari
- Twitter: http://twitter.com/cdesio
- LinkedIn: http://www.linkedin.com/in/claudiodesio
- Google +: http://plus.google.com/+ClaudioDeSioCesari
- Internet: http://www.claudiodesio.com

In questo modo otterrete aggiornamenti su materiale on line come errata corrige, nuove versioni di appendici ed esercizi, pubblicazioni, iniziative e così via. È anche possibile contattare l’autore tramite email a claudio@claudiodesio.com, per feedback, recensioni, chiarimenti, domande e richieste.

Indice analitico

Simboli

!= (diverso da)
%= (modulo ed assegnazione)
& (AND)
&= (AND e assegnazione)
*= (moltiplicazione ed assegnazione)
++ (pre-incremento di un’unità)
+= (somma ed assegnazione)
-- (pre-decremento di un’unità)
-= (sottrazione ed assegnazione)
-disableassertions
-enableassertions
-Xlint
/= (divisione ed assegnazione)
< (minore)
<< (shift a sinistra)
<<= (shift a sinistra e assegnazione)
<= (minore o uguale)
== (uguale a)
> (maggiore)
>= (maggiore o uguale)
>> (shift a destra)
>>= (shift a destra e assegnazione)
>>> (shift a destra senza segno)
>>>= (shift a destra senza segno e assegnazione)

`©interface` (parola chiave)
`\n`
`\r`
^(XOR)
^= (XOR e assegnazione)
| (OR)
|= (OR e assegnazione)
~ (NOT)

A

`abstract` (parola chiave)
Accessor methods
Accoppiamento
Algoritmo
Annotazioni
  a valore unico
  di tipi
  marcatrice
  standard
Apertura qualificata
APT (Annotation Processing Tool)
`args[]`
Array
`arrayCopy()`
`Arrays` (classe)
`assert` (parola chiave)
Asserzioni
Astrazione
Attributo
Autoboxing - autounboxing

B

`BlockingQueue` (interfaccia)
`Boolean, boolean` (parola chiave)
`break` (parola chiave)
Byte Stream

C

Carriage return
Cast
Character Stream
Ciclo for migliorato
`Class` (classe)
`ClassCastException`
Classe
- anonima
- innestata
- Optional
- wrapper

`clone()`
Closures
Code FIFO e LIFO
Coesione
`Collection` (interfaccia)
`Collections` (classe)
Collezioni eterogenee
Compact Strings
`Comparable` (interfaccia)
`Comparator` (interfaccia)
Complemento a due
Concorrenza

Convenzioni
Copie immutabili
Costruttore
`currentTimeMillis()`
`CyclicBarrier` (classe)

D

Date-Time API
Date classes
Decorator (pattern)
Deduzione automatica del tipo
deep copy
delimitatore del token
`Deprecated` (annotazione)
`Deque` (interfaccia)
descrittore del modulo
Diamond Problem
DIP (Dependency Invertion Principle)
documentazione Java
`Documented` (annotazione)
`Duration` (classe)

E

Eccezioni
  checked e unchecked
Eclipse
EJE,
Enumerazioni
  `ChronoUnit`
  `DayOfWeek`

`Month`
Enumerazioni, `enum` (parola chaive)
`switch`
`equals()`
Ereditarietà
multipla
`Error`
Espressioni Lambda
Espressioni lambda
Espressioni regolari
Executors
`exports` (parola chiave)
`extends` (parola chiave)

F

Factory Method (pattern)
`Files` (classe)
`final` (parola chiave)
`finally` (parola chiave)
Foreach
Fork/Join
Forwarding reference (JShell)
Framework Collections
FunctionalInterface (annotazione)

G

Garbage Collector
Generalizzazione
Generici (Generics)
Generic type

Geolocalizzazione

H

`hashcode()`
`HashMap` (classe)
`HashSet` (classe)
`Hashtable` (classe)
HelloModularWorld
HelloWorld

I

i18n
Immutabile
Implementare
`implements` (parola chiave)
`import` (parola chiave)
`Inherited` (annotazione)
Inizializzatori
  d’istanza
  statici
`instanceof` (parola chiave)
`Instant` (classe)
Internazionalizzazione
Invariante
Istanziare
`Iterator`
iteratori

J

JAR

Java
Javadoc
Jlink
JMOD
JShell

L

Lint
`List` (interfaccia)
Literals
`LocalDateTime` (classe)
`Locale` (classe)
`LocalTime` (classe)
Lock

M

`Map` (interfaccia)
`Math` (classe)
Memoria
  gestione
Memory leak
Meta-annotazioni
Meta-programmazione
Metadati
Metodi detti di convenienza
Metodi sincronizzatori
Metodo
  dichiarazione
  generico
  helper

Migrazione di programmi pre-Java
Modulepath
Moduli, `module` (parola chiave)
Moduli automatici
Modulo anonimo
Monitor
Multi catch
Mutator methods

N

`NaN`
`Native` (annotazione)
`native` (parola chiave)
`NavigableMap` (interfaccia)
`NEGATIVE_INFINITY`
Netbeans
New line
NIO
Node stream
`notify()`
`notifyAll()`

O

`Object` (classe)
OCL (Object Constraint Language)
Oggetto
  dichiarazione
`open` (parola chiave)
`opens` (parola chiave)
Operatore ternario

Operatori
  bitwise
  logico - booleani
Operatori aritmetici
Operatori di incremento
Operazione di riduzione
Operazione terminale
Operazioni di aggregazione
Ordinamento
Overload
Override
`Override` (annotazione)

P

Package
ParallelGC
Parameter type
Parametri
  bounded
  covarianti
  polimorfi
Parametri formali
Partizionamento del software
Partizionamento orizzontale
Partizionamento verticale
`Path` (interfaccia)
`Period` (classe)
Permits
Pipeline
Polimorfismo

`POSITIVE_INFINITY`
Postcondizione
Precondizione
Preemptive scheduling
`private` (parola chiave)
Progettazione per contratto
Promozione (promotion)
Propagazione dell’eccezione
`protected` (parola chiave)
`provides` (parola chiave)
`public` (parola chiave)

Q

`Queue` (interfaccia)

R

Raw type
Reference
Reflection
`Repeatable` (annotazione)
`requires` (parola chiave)
`ResourceBundle` (classe)
`Retention` (annotazione)
Riuso
Round-robin scheduling
`Runnable` (interfaccia)
`Runtime`

S

SAM (Single Abstract Method)
Scheduler
`Semaphore` (classe)
ServiceLoader
`Set` (interfaccia)
Shallow copy
Sincronizzazione
Singleton
Sistema numerico
`SortedMap` (interfaccia)
`SortedSet` (interfaccia)
specializzazione
`static` (parola chiave)
  import
Stream API
`strictfp` (parola chiave)
StringTokenizer (classe)
Strong encapsulation (forte incapsulamento)
`super` (parola chiave)
`SuppressWarnings` (annotazione)
`synchronized` (parola chiave)
`System` (classe)
  properties

T

`Target` (annotazione)
Temporal Adjusters
`TemporalQueries` (classe)
`this` (parola chiave)
Thread

multithreading
`throw` (parola chiave)
`throws` (parola chiave)
Time slicing
Tipi senza segno
`toString()`
`transient` (parola chiave)
`transitive` (parola chiave)
`TreeMap` (classe)
`TreeSet` (classe)
Try with resources
Type erasure

U

UML
Underscore
Unicode
Unsigned
URI
`uses` (parola chiave)

V

Varargs
Variabile
d’istanza
locali
Variabili implicite (JShell)
`volatile` (parola chiave)

W

`wait()`
Warnings
Wildcard
  bounded
  capture
  unbounded

X

XLint

Informazioni sul Libro

Java non sarà mai più lo stesso!

Il *Manuale di Java 9* è stato strutturato per soddisfare le aspettative di

Aspiranti programmatori: nulla è dato per scontato, è possibile imparare a program-mare partendo da zero ed entrare nel mondo del lavoro dalla porta principale.

Studenti universitari: le precedenti edizioni di quest'opera sono state adottate come libro di testo per diversi corsi in tutte le maggiori università italiane.

Programmatori esperti: Java 9 semplifica lo sviluppo, rivoluziona il JDK, introduce nuovi formati per i file e, con i moduli, cambia per sempre il modo in cui pensiamo, progettiamo e distribuiamo il software.

Circa l’autore

Claudio De Sio Cesari (http://www.claudiodesio.com) è consulente free lance specialista in formazione, sviluppo, analisi, progettazione, architettura e metodologie object oriented. Con Hoepli ha già pubblicato tre edizioni di grande successo del *Manuale di Java*. Ha collaborato con diverse Università, Enti ministeriali e aziende tra cui Sun Microsystems (ora Oracle) creatrice di Java, dove è stato per diversi anni trainer e mentor.